

NAVAL POSTGRADUATE SCHOOL MONTEREY, CALIFORNIA



THESIS

USING THE PEBB UNIVERSAL CONTROLLER TO
MODIFY CONTROL ALGORITHMS FOR
DC-TO-DC CONVERTERS AND IMPLEMENT
CLOSED-LOOP CONTROL OF ARCP INVERTERS

by

David L. Floodeen

September 1998

Thesis Co-Advisors

John G. Ciezki
Robert W. Ashton

Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED 4

19981113 084

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE September 1998	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE USING THE PEBB UNIVERSAL CONTROLLER TO MODIFY CONTROL ALGORITHMS FOR DC-TO-DC CONVERTERS AND IMPLEMENT CLOSED-LOOP CONTROL OF ARCP INVERTERS		5. FUNDING NUMBERS	
6. AUTHOR(S) David L. Floodeen			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		12b. DISTRIBUTION CODE	
13. ABSTRACT (maximum 200 words) <p>The objective of this thesis is two-fold. The first goal is to expand the operational capabilities of the Ship's Service Converter Module control algorithm for a DC-to-DC converter using the Universal Controller. The second goal is to investigate the use of the Universal Controller to implement a closed-loop control algorithm for an Auxiliary Resonant Commutated Pole (ARCP) power inverter. These power electronic devices are central to the development of a DC Zonal Electric Distribution System (DC ZEDS) that is scheduled for application in the twenty-first century surface combatant (SC-21). The development of appropriate control algorithms is a key element to this design process. The Universal Controller is a digital controller that was developed by personnel at the Naval Surface Warfare Center (NSWC), Annapolis, Maryland. The basic operation of the Universal Controller and the Texas Instrument TMS320C30 microprocessor architecture are described, with emphasis placed on the system control algorithms.</p> <p>Previous studies have encoded and successfully tested a closed-loop control algorithm for a DC-to-DC converter. In this research endeavor, this control algorithm is expanded to include various protection circuits and a Master/Slave paralleling scheme. Finally, a closed-loop control algorithm for the ARCP inverter is encoded and recommendations for future research are outlined.</p>			
14. SUBJECT TERMS dc-to-dc buck converter, auxiliary resonant commutated pole inverter, universal controller, closed-loop control of power inverters, texas instruments tms320c30		15. NUMBER OF PAGES 124	
		16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18 298-102

Approved for public release; distribution is unlimited.

**USING THE PEBB UNIVERSAL CONTROLLER TO MODIFY CONTROL
ALGORITHMS FOR DC-TO-DC CONVERTERS AND IMPLEMENT
CLOSED-LOOP CONTROL OF ARCP INVERTERS**

David L. Floodeen
Lieutenant Commander, United States Navy
B.S.E.E., San Diego State University, 1987

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
September 1998**

Author:



David L. Floodeen

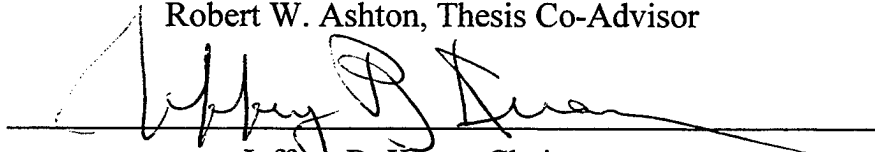
Approved by:



John G. Ciezki, Thesis Co-Advisor



Robert W. Ashton, Thesis Co-Advisor



Jeffrey B. Knorr, Chairman

Department of Electrical and Computer Engineering

ABSTRACT

The objective of this thesis is two-fold. The first goal is to expand the operational capabilities of the Ship's Service Converter Module control algorithm for a DC-to-DC converter using the Universal Controller. The second goal is to investigate the use of the Universal Controller to implement a closed-loop control algorithm for an Auxiliary Resonant Commutated Pole (ARCP) power inverter. These power electronic devices are central to the development of a DC Zonal Electric Distribution System (DC ZEDS) that is scheduled for application in the twenty-first century surface combatant (SC-21). The development of appropriate control algorithms is a key element to this design process. The Universal Controller is a digital controller that was developed by personnel at the Naval Surface Warfare Center (NSWC), Annapolis, Maryland. The basic operation of the Universal Controller and the Texas Instrument TMS320C30 microprocessor architecture are described, with emphasis placed on the system control algorithms.

Previous studies have encoded and successfully tested a closed-loop control algorithm for a DC-to-DC converter. In this research endeavor, this control algorithm is expanded to include various protection circuits and a Master/Slave paralleling scheme. Finally, a closed-loop control algorithm for the ARCP inverter is encoded and recommendations for future research are outlined.

TABLE OF CONTENTS

I. INTRODUCTION	1
A. DC ZONAL ELECTRICAL DISTRIBUTION	1
B. RESEARCH FOCUS	3
II. UNIVERSAL CONTROLLER	7
A. INTRODUCTION	7
B. GENERAL DESCRIPTION	8
C. PRIMARY COMPONENTS.....	9
1. CPU Board	9
2. I/O Board.....	11
D. OPERATIONAL OVERVIEW	14
III. TMS320C30 ARCHITECTURE.....	17
A. INTRODUCTION	17
B. ARCHITECTURE.....	17
C. ADDRESS MODES.....	19
D. PROGRAM DEVELOPMENT AND SUPPORT	20
IV. BUCK CHOPPER APPLICATIONS	23
A. INTRODUCTION	23
B. PROTECTION CIRCUITS.....	25
1. 24 volt Control Power Low/Over-Temperature	25
2. Over-Current Sense and Shutdown	26
C. LOCAL/REMOTE SWITCH MODIFICATION	29
1. Specifications.....	29
2. Application	29
D. MASTER/SLAVE PARALLELING.....	33
1. Theory	33
2. Application	35
3. Findings	37
V. C PROGRAMMING ISSUES.....	39
A. INTRODUCTION	39
B. C INTERFACE REQUIREMENTS	40
C. PROBLEMS WITH IMPLEMENTATION IN EXISTING CODE	41
VI. ARCP CONTROL.....	45
A. BASIC ARCP INVERTER OPERATION.....	45
B. OPEN-LOOP CONTROL.....	46
C. CLOSED-LOOP CONTROL	51
1. Theory	51
2. Application	54
VII. CONCLUSIONS.....	55
A. SUMMARY OF RESEARCH WORK	55
B. NOTABLE CONCLUSIONS	56
C. RECOMMENDATIONS FOR FUTURE WORK	56
APPENDIX A. SOFTWARE ACCESS AND DOS COMMANDS.....	59
A. PROGRAM DEVELOPMENT SOFTWARE TOOLS.....	59

B. DOS COMMANDS	59
C. BATCH FILES	62
APPENDIX B. BUCK CHOPPER CONTROL CODE.....	65
APPENDIX C. ARCP CLOSED-LOOP CONTROL CODE.....	93
LIST OF REFERENCES	113
INITIAL DISTRIBUTION LIST	115

I. INTRODUCTION

A. DC ZONAL ELECTRICAL DISTRIBUTION

Downsizing is a reality in the military of today. The United States Navy is continually tasked with finding ways to meet operational commitments as well as satisfy the research and development requirements needed to continually upgrade capabilities. One area the Navy is investigating is the use of a DC power distribution system for the next generation of ships. The project is referred to as DC Zonal Electrical Distribution System (DC ZEDS) [Ref. 1]. Figure 1-1 shows a simplified block diagram of a proposed DC ZEDS system.

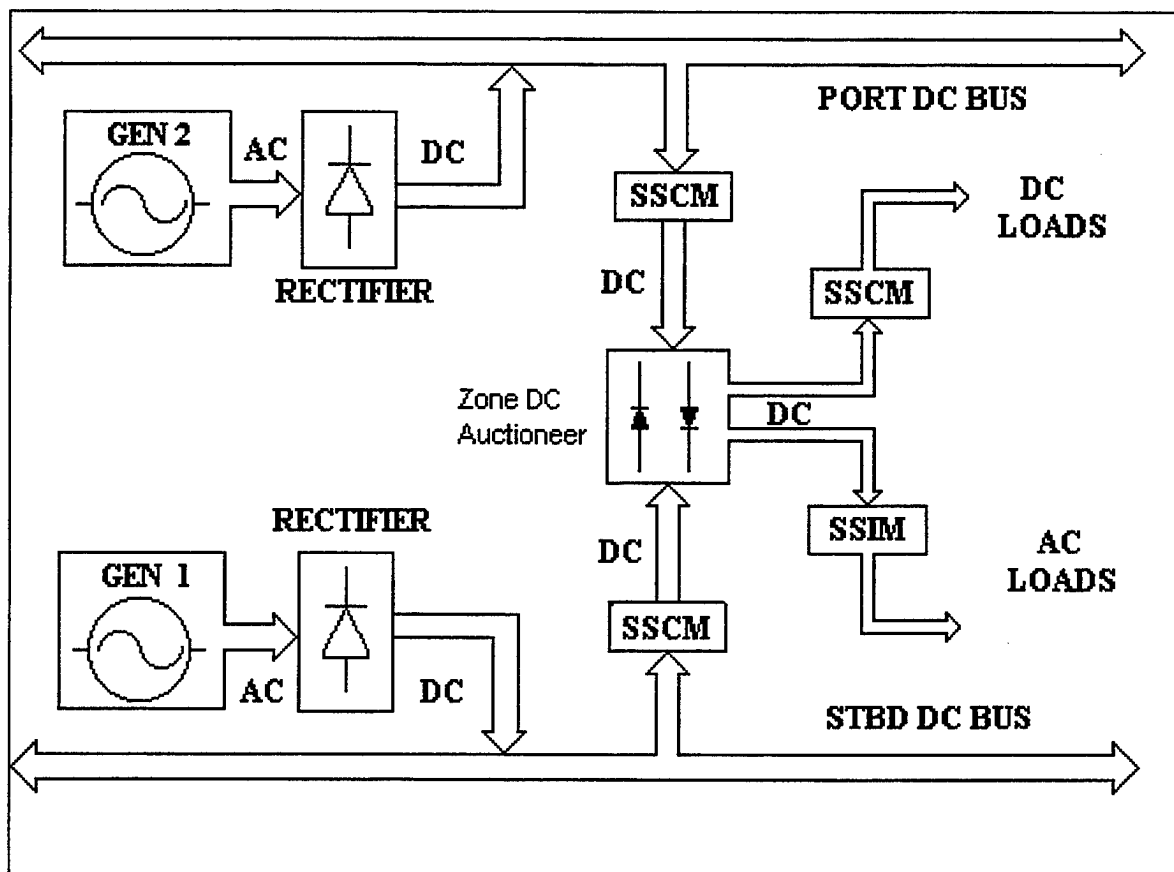


Figure 1-1 DC Zonal Electrical Distribution System [Ref. 2]

Power distribution under this system is accomplished by rectifying AC into DC as soon as it is generated. The ship is divided into zones and the DC power is routed to these zones on two primary DC busses (Port DC bus and Stbd DC bus). Upon entering the zone, the DC power is stepped down using Ship's Service Converter Modules (SSCMs) that act as buffers for the zones. The power is then further stepped down using more SSCMs or converted back to AC using Ship's Service Inverter Modules (SSIMs). DC power distribution can increase survivability by speeding up the fault detection and switching process, and because DC ZEDS requires significantly less cabling and essentially no transformers, it is projected to produce large savings in both the weight and the cost of next generation ships. [Ref. 1]

The SSCMs are actually buck chopper converters that are used to step down and regulate the DC voltage entering the zone. The SSIMs are Auxiliary Resonant Commutated Pole (ARCP) inverters that convert the DC into three-phase (3 ϕ) AC [Ref. 2]. Both SSCMs and SSIMs require feedback control and monitoring to be useful in a DC ZEDS environment because the systems must be stable and allow for fast transient response in the dynamic environment aboard U.S. Navy ships. Digital control has proven to be more flexible due to the ability to modify the control algorithm with simple software changes vice the extensive hardware changes required in analog systems. For DC ZEDS to be successful, effective control algorithms for SSCMs and SSIMs must be developed.

B. RESEARCH FOCUS

The focus of this thesis is on using the Power Electronic Building Blocks (PEBB) Universal Controller, developed by the engineers at Naval Surface Warfare Center (NSWC), to expand the operational capabilities of the SSCM control algorithm and to implement a closed-loop control algorithm for the SSIM. The PEBB Universal Controller is a two-card digital controller designed to handle the extensive Input/Output (I/O) requirements needed to control both buck chopper converters and ARCP inverters. Closed-loop control for a typical ARCP inverter requires the conversion of as many as 10 voltage and current signals and the generation of as many as 12 different control signals. These control signals are used to gate on and off the electronic switches and modify the duty cycle. The Universal Controller has no user's manual. Previous research [Ref. 3] documented, in part, how the Universal Controller works and how to implement control algorithms using it. Chapter II of this thesis documents in greater detail the actual operation of the Universal Controller.

The Universal Controller is based on the Texas Instrument TMS320C30 microprocessor. This is a general purpose microprocessor designed for DSP applications. Chapter III delves into the architecture of this chip and how it gives flexibility to the Universal Controller.

Previous research [Ref. 3] has included the encoding and successful testing of a closed-loop control algorithm for the SSCM, the buck chopper. Several additional operational features were desired by NSWC. Over-current protection, under-voltage protection, over-temperature protection, and the ability to operate the bucks from a remote front panel were all desired features that were not incorporated in the original

buck control code. Chapter IV of this thesis contains a discussion of the theory and changes required to implement these features.

The present SSCM closed-loop control algorithm implementation [Ref. 3] allowed for individual control of multiple buck choppers. Problems developed when trying to operate these units in parallel at Power Paragen, Inc., Anaheim CA. One buck had a tendency to take over and try to supply the entire load while the other unit floated at no load. The changes made to operate the buck chopper converters in a Master/Slave configuration that provides the proper current sharing required for successful parallel operation are documented and discussed in Chapter IV as well.

The TMS320C30 has the ability to be programmed in both assembly language and C. Closed-loop control algorithms use complex mathematical functions to calculate the desired control signals. The current encoded control algorithms are written in assembly language. This code is lengthy and very complex. Great benefits would be derived from using C code functions to implement the control algorithms. C code, being a high-level language, uses instructions that more closely resemble common mathematical statements. Using C would greatly improve the readability of the software which would, in turn, facilitate easier modifications. The possibility of converting parts of the existing program to C is investigated and reported on in Chapter V.

NSWC engineers encoded an open-loop control algorithm for the SSIM, an ARCP inverter. Closed-loop control is desirable because it can reduce or eliminate changes that would occur in output voltages caused by changes in the load or input voltage [Ref. 4]. Chapter VI of this thesis contains a description of the current open-loop operation of the ARCP and the implementation of one proposed closed-loop control

algorithm. Finally, Chapter VII contains a summary of research work completed, notable conclusions, and recommendations for future work.

II. UNIVERSAL CONTROLLER

A. INTRODUCTION

Digital control algorithms have proven more flexible than analog ones. Changes to digital controllers can be made relatively easily via software modifications. Analog changes require the removal and replacement of actual components. This can be very time consuming and expensive. Also, depending on component tolerances, the accuracy of the analog implementation may be less than acceptable. Digital algorithms, on the other hand, can be modified by changing numbers in software then reloading the new program. Any size change can be accommodated with the proper scaling, and accuracy can be achieved by "fine tuning" the changes in the software.

Closed-loop control algorithms can be very I/O intensive. The ARCP, for example, has six (6) primary switches that require control signals to turn on and off the solid-state gates. Also, 3-phase power control implies 3 different phase current and voltage measurements that need to be sampled and manipulated. Finding an appropriate digital controller that can handle this many I/O signals is challenging. For this reason, the engineers at NSWC designed the Power Electronic Building Block (PEBB) Universal Controller, here in referred to as simply the Universal Controller. PEBB is a generic term for solid-state switching equipment being developed for Department of Defense (DOD) systems and Universal Controller implies that this controller is designed to handle a myriad of applications in addition to those discussed in this thesis.

B. GENERAL DESCRIPTION

The PEBB Universal Controller is comprised of two basic parts, a CPU board and an I/O board. Figure 2-1 shows a block diagram of the Universal Controller. The CPU board is based on the Texas Instruments TMS320C30 DSP microprocessor chip which will be covered in greater detail in Chapter III. The CPU board also contains three (3) different types of memory and a microcontroller that directs the interface with the host PC. The I/O board contains the Analog-to-Digital (A/D) converters that provide the analog input to the Universal Controller and has several counter/timers used to generate interrupts and modify the output control signals from the board.

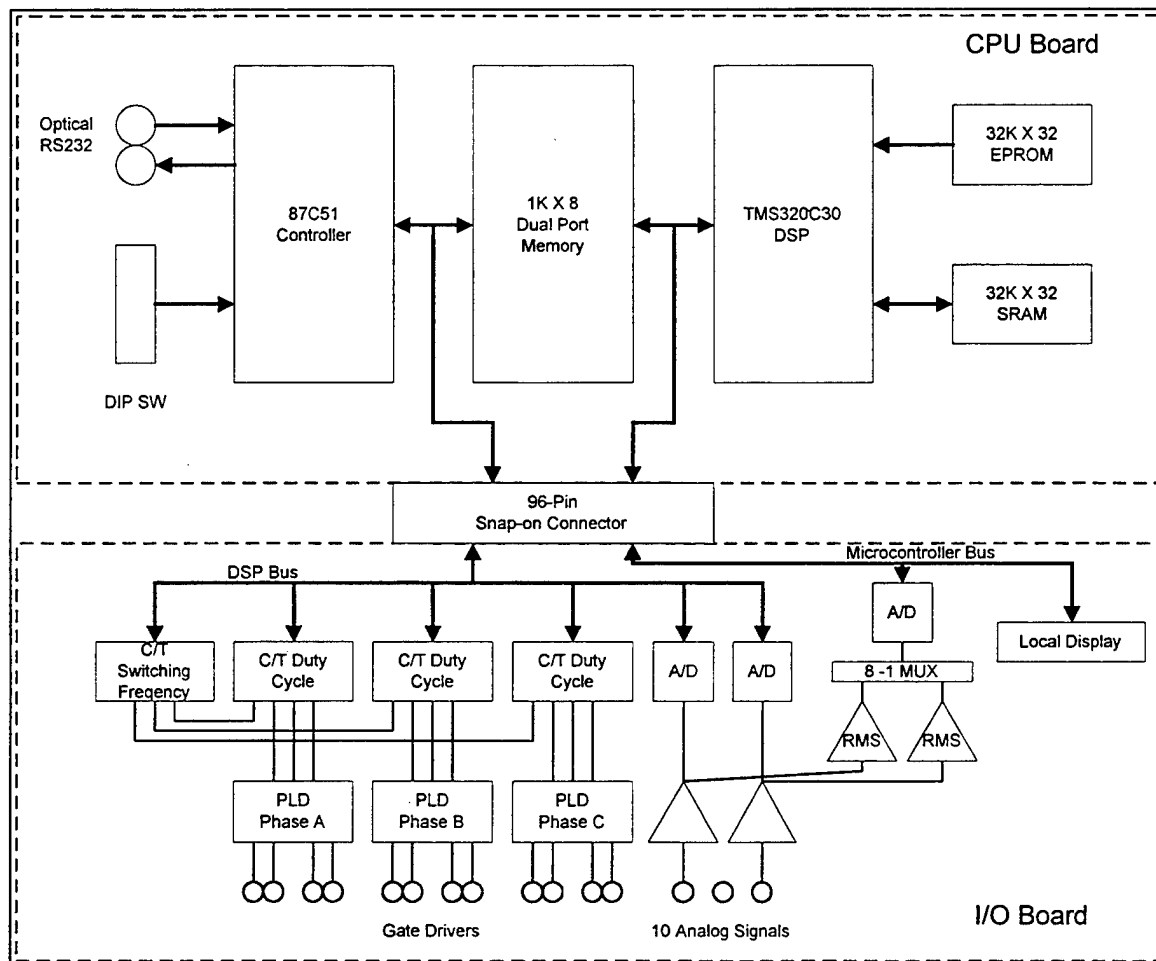


Figure 2-1 PEBB Universal Controller Block Diagram [Ref.5]

Input to the Universal Controller comes from a host PC using the RS232 serial port on the back of the PC. Output from the Universal Controller is converted to optical gate control signals by optical transmitters on the I/O board. These signals are then sent out as control signals to modify switch operation. This provides a level of isolation between the Universal Controller and the high-power units being controlled.

C. PRIMARY COMPONENTS

1. CPU Board

The CPU board is built around the Texas Instrument TMS320C30 which, as stated earlier, is discussed in the next chapter. The microprocessor is supported by the Texas Instrument TI 8751 microcontroller and a memory section. Figure 2-2 shows the primary components of the Universal Controller CPU board.

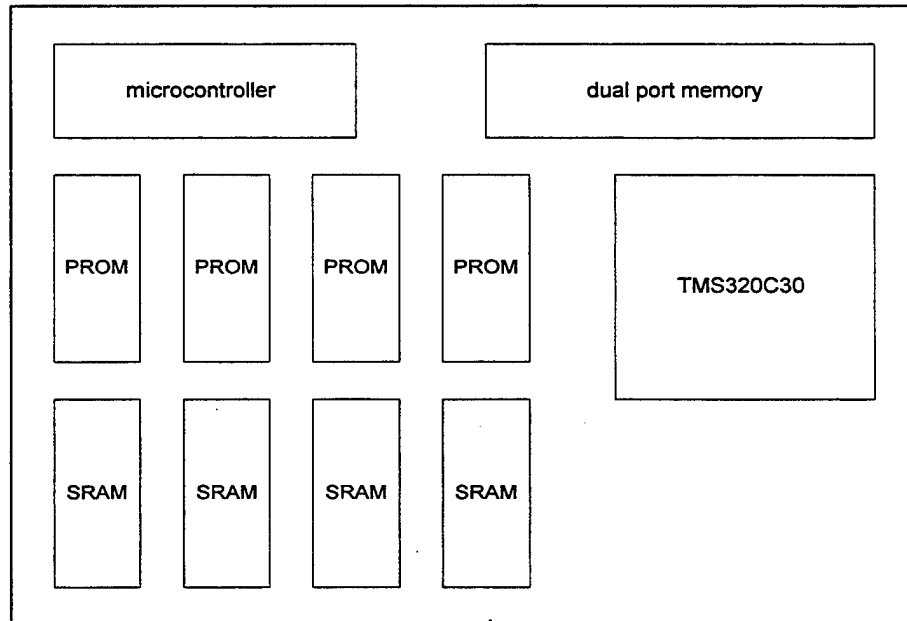


Figure 2-2 CPU Board of the Universal Controller [Ref. 3]

The microcontroller is able to communicate both in serial or parallel modes. Any one of its 32 I/O pins can be addressed as an input, an output, or both. [Ref. 6] This

is what gives the TI 8751 the flexibility to communicate serially with the host PC and yet read from and write to the dual port memory in parallel.

The TI 8751 microcontroller also contains 4K bytes of Erasable Programmable Read Only Memory (EPROM) on-chip. This is used to hold the interface program supplied by NSWC. A copy of this code is located on PCPWR7, a personal computer located in Bullard Hall room 114. This code is what controls the interface between the Universal Controller and the host PC. It contains the memory map and instructions used for loading front panel information from the PC into the Universal Controller on start-up and it also directs the interrupts generated by the host PC used to initiate and terminate operation of the controller.

The second major portion of the CPU board is the memory section. The memory section can be divided into three parts. The three types of memory located on the Universal Controller are 32K of EPROM, 32K of Static Random Access Memory (SRAM), and 1K x 8-bit high-speed dual port static RAM. The EPROM part of memory is made by connecting four WSI WS57C256F 32K x 8-bit chips in parallel. Since the EPROMS have an 8-bit data word and the TMS320C30 uses a 32-bit data bus, an address decoder is connected to the four WSI EPROMS. This allows simultaneous access to the four memory chips and provides a 32-bit data word for the CPU. The EPROM memory is used primarily for storage of the operating program for the Universal Controller.

The static RAM, or SRAM, is made up of four IDT71256SA fast 32K x 8-bit CMOS chips. Again, these chips are connected in parallel to an address decoder to provide a 32-bit data word similar to the EPROMs. The SRAM is primarily used for

data storage. It stores values such as the sin look-up table used by the ARCP program for calculating the control signals.

The final type of memory is the 1K x 8-bit high-speed dual port static RAM. It is connected between the microcontroller and the microprocessor. This memory is used to store information sent to the Universal Controller from the host PC until it is needed or until it can be loaded into the SRAM. This information includes, but is not limited to, command information that directs which algorithm to run, maximum and minimum currents and voltages, reference information used by the control algorithm, and the control constants needed by the control equations. Because the Universal Controller communicates serially with the host PC, access to this information is delayed a relatively long time. Using the microcontroller to direct this interface and load this information into the dual port memory for later use by the microprocessor greatly accelerates this process. A more in-depth discussion of the specifics of the Universal Controller's memory is available in Chapter III of Reference 3.

2. I/O Board

The I/O Board of the Universal Controller can be divided into two main functional parts, an analog-to-digital interface portion and a counter/timer portion. Figure 2-3 shows the key components of the I/O board of the Universal Controller.

The analog-to-digital interface part consists of 11 A/D converters used to convert voltage and current signals into digital words that can be used by the microprocessor. The counter/timer portion of the I/O board is made up of 4 counter/timer chips that contain 3 counters each and are used for various timing applications needed for control algorithm implementation.

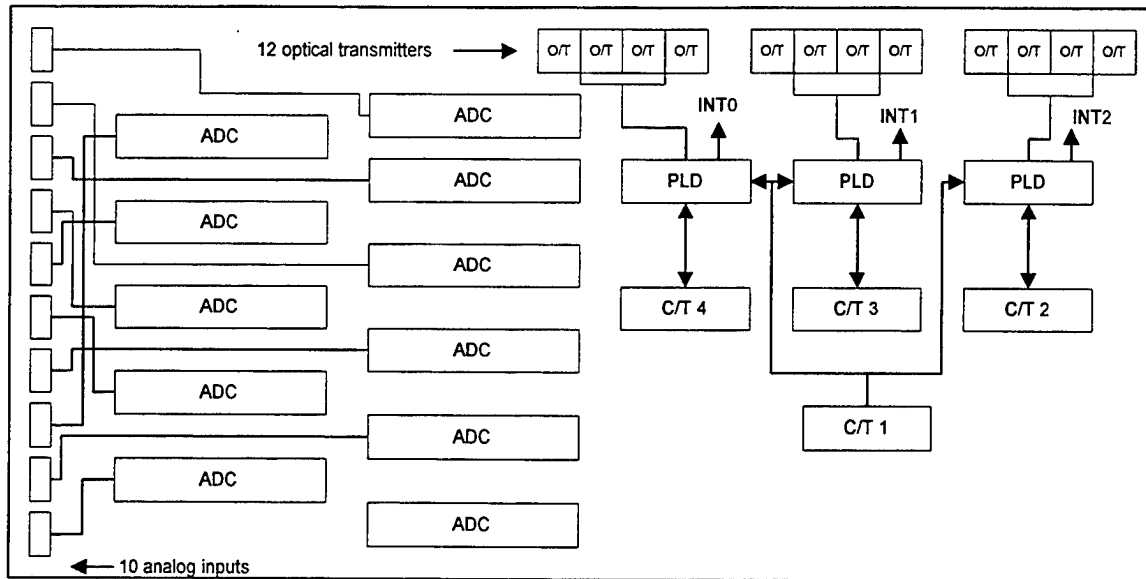


Figure 2-3 I/O Board of the Universal Controller [Ref. 3]

The analog-to-digital interface is made up of 11 Maxim 500kbps A/D converters. Ten of these 12-bit converters are used to convert sensed voltages and currents that are imported to the control board from the SSCMs or SSIM. One is reserved for conversions of on-board values needed for different types of operation. Five memory locations are reserved for the A/D converters. Since the TMS320C30 uses 32-bit words, two (2) 12-bit A/D words can be stored in each location. Initiating a read of an A/D converter's memory location will read the value of the last conversion and initiate a new one. The first read is always discarded because it is the result of a previous, or time late, data sample. The second read represents a more real-time sample of the desired data and is saved for computations. Each conversion takes 2.6 μ sec which is considerably slower than the 60 nsec instruction execution time for the microprocessor. To allow for complete conversions, there is a wait loop or delay time programmed into the code

following each A/D read. Specifics on data step size and digital word selectivity of the A/D converters can be found in Reference 6.

Four (4) Harris 82C54 counter/timers are located on the I/O board. Each 82C54 contains 3 counters that can be set up in various modes of operation for use by the Universal Controller. The first counter/timer is operated in a rate generator mode. This counter/timer functions as the switching frequency timer. The desired switching frequency is programmed from the PC and converted to a count based on the clock speed and then loaded into the counter.[Ref. 3] The other two counters located on the same chip are loaded with the same switching frequency count only delayed either by 120 degrees or 180 degrees of switching period, depending on the application. At a 20 Khz switching frequency, 120 degrees of delay equates to 16.7 msec and 180 degrees of delay is 25 msec of difference between the first counter and the next. These counters will generate interrupts with the proper phase shifts needed by the Universal Controller to initiate sampling and run the control algorithms.

The other three (3) Harris counter/timers are used as hardware retriggerable one-shots. Adjusting the count in these three (3) counters changes the duty cycle of the controlled solid-state switches. The duty counts calculated by the control equations are loaded into these counters. The output pulses of these three counters go to optical transmitters via three (3) PALs that convert the timer outputs into the necessary control signals for the SSCMs or SSIM switches. Further details on the actual loading of these counter/timers and count calculation details can be found in Reference 7.

D. OPERATIONAL OVERVIEW

The process of operating the Universal Controller can be divided into several steps. The EPROMS are programmed, loaded into the Universal Controller CPU board, and power is applied. The next step is to load the operation and control values from the host PC and begin operation. Once the program is running, phase interrupts generated by the switching frequency timer will run the control algorithms. The Universal Controller will continue operating in this interrupt driven mode until the unit is shut down.

Operation begins with loading the operating code for the microprocessor onto the EPROM's. To do this, the code is assembled, linked, converted to the proper format and "burned" into the PLDs. The assembler is loaded on PCPWRP-8 located in Bullard Hall room 114. This is an older PC that uses DOS 3.x as an operating system. This machine is still used to assemble the code because it also has the ALL-03A Universal Programmer and Tester attached to it which programs the WSI PLDs. This allows the code to be assembled and loaded onto the PLDs all on the same system. To make the task of programming chips easier, batch files have been created. A batch file is a DOS file that contains a listing of executable instructions. To use a batch file, simply type the name of the file and it will execute the necessary instructions. Appendix A contains a listing of DOS commands and instructions for creating, modifying, and using batch files to assemble the code and load the executable object file onto PLDs.

Once the code is loaded, the four (4) PLDs are inserted into the four (4) PROM slots [Fig. 2-2] as U5, U6, U8, and U9 on the CPU board. Then, when power is applied to the Universal Controller, the software program initializes the microprocessor, the

memory map, the counter/timers, and the interrupt structure and then waits for a “unit on” interrupt (interrupt 3 in the code) from the host PC. Reference 3 describes in detail the operation of the host PC and associated software. Figure 2-4 illustrates the program flow. The “unit on” interrupt starts the operation of the control algorithms. The front panel values are read and loaded into the dual port memory, the counter/timers (C/Ts) are loaded, the rest of the interrupts are enabled and the unit then waits for a phase interrupt to occur.

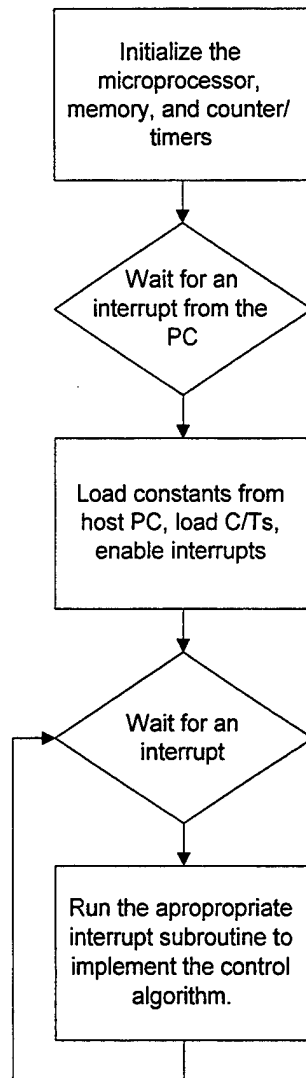


Figure 2-4 Program Flow for the Universal Controller

Interrupts drive the system. The interrupts occur with the correct phase relationship, and at the proper sampling rate determined by the switching frequency loaded in from the host PC. The interrupt subroutines sample the required voltages and currents; the microprocessor manipulates the data and calculates the duty cycle changes needed to produce the proper outputs. The counts loaded into the counters/timers that control the switching period of the IGBTs are modified. The outputs of these counters produce the control signals that are converted to optical signals and sent out to the SSCMs or SSIM. Again, all of these actions are controlled by the program run by the TMS320C30 microprocessor. The architecture of the TMS320C30 is discussed in the next chapter.

III. TMS320C30 ARCHITECTURE

A. INTRODUCTION

The TMS320C30 is the heart of the Universal Controller. It is a high-speed general-purpose microprocessor produced by Texas Instruments. It has an architecture, instruction set, and support system conducive to real-time digital signal processing (DSP) and ideal for application as the center piece of the Universal Controller. The TMS320C30 has a 60 nsec single cycle execution time that gives it the speed to execute up to 50 MFLOPS [Ref. 8]. Many functions that are often done with software are performed in hardware by the C30. This architecture allows a high level of parallelism to support pipelining which increases speed. The TMS320C30 supports multiple addressing modes. Six different types of addressing are available in five different modes. This gives the C30 the large amount of versatility needed to implement complex control algorithms. Texas Instruments provides several support programs with extensive documentation to aid in system development using the TMS320C30. It is this high-speed architecture, the flexible addressing modes, and the extensive support systems that makes the TMS320C30 ideal for use in the PEBB Universal Controller.

B. ARCHITECTURE

The TMS320C30 uses a register-based architecture. It consists of 12 control registers, 8 extended precision registers (also called accumulators) and 8 auxiliary registers. This register system give the C30 the flexibility to handle complex tasks using registers for storage. By decreasing the number of times the CPU needs to access

memory, the overall speed of the system is increased. The TMS320C30 also contains two auxiliary register arithmetic units (ARAU) that are used strictly for address calculations. The ARAUs can generate two (2) different addresses in a single clock cycle [Ref. 8]. They are used to calculate complex addresses such as addresses with displacement or addresses used in the circular addressing mode which are discussed in a later chapter. Being able to separately and simultaneously calculate memory addresses allows a great deal of pipelining. Pipelining is the overlapping of instructions being executed by the microprocessor which greatly increases speed of operation. In the case of memory access, using ARAUs to calculate memory addresses frees up the ALU to perform other tasking while the C30 is reading from or writing to memory. An added benefit of using ARAUs comes from freeing the other microprocessor registers, which would normally be used for memory address calculations, to be used as needed elsewhere in the program.

In addition to the flexible register system, the TMS320C30 has other pieces of hardware that add to its overall speed. The C30 has a full function ALU that performs operations on 32-bit integers and 40-bit floating point data in a single clock cycle. There is also a Barrel shifter capable of performing up to 32-bit shifts in a single cycle which adds great flexibility for bit manipulation instructions. Finally, the TMS320C30 has a parallel floating point/integer multiplier. This multiplier allows floating point operations to be performed in parallel with ALU operations. The inputs to the multiplier are two (2) 32-bit floating point numbers and the result is a 40-bit floating point number [Ref. 8]. The instruction set of the TMS320C30 is written to support parallel instruction execution so programs can easily be written to take advantage of this

parallel architecture. Simultaneous use of the single cycle ALU, Barrel shifter, and parallel multiplier are possible in software [Ref. 8].

The registers, the ALU, and the parallel multiplier are all supported by an extensive 32-bit internal bus structure designed to allow a great deal of instruction overlap in execution. This bus structure is what enables the parallel instruction set. In addition to two (2) ARAU address buses and two(2) separate data busses that connect CPU registers to memory, another set of separate address and data busses are used for peripherals and yet another for Direct Memory Access (DMA). It is this extensive bus structure supporting the large amount of paralleling hardware and the flexible register system that makes the TMS320C30 well suited for use in the Universal Controller.

C. ADDRESS MODES

Much of the flexibility of the TMS320C30 comes from the instruction set that supports it. This instruction set is quite powerful due in part to the numerous addressing modes available for use. The C30 supports five(5) different addressing modes and six(6) types of addressing. Table 3-1 shows a listing of the different addressing modes and types used by the TMS320C30.

Five Addressing Modes	Six Addressing Types
General Addressing Modes	Register Addressing
Three-Operand Addressing Modes	Direct Addressing
Parallel Addressing Modes	Indirect Addressing
Conditional Addressing Modes	Short-Immediate Addressing
Circular Addressing Modes	Long-Immediate Addressing
	PC-Relative Addressing

Table 3-1 TMS320C30 Addressing Modes and Types

An addressing mode is a grouping of instructions based on the syntax used when writing code. An addressing type is a grouping of instructions based on how data is accessed from memory or registers. Chapter 5 of Reference 8 provides a complete and thorough description of each mode and type of addressing. Not every type of addressing is available in every mode. For instance, the Three-Operand Addressing Mode allows only register addressing and indirect addressing because of the fields available in the instruction word. Examples of each mode and type of addressing can be found in Appendix B (the code). Circular addressing plays a very important role in the implementation of the control algorithms discussed in Chapter VI and will be covered in more detail there.

D. PROGRAM DEVELOPMENT AND SUPPORT

Texas Instruments provides an excellent support system for the TMS320C30 microprocessor user. Numerous resources are available to aid in the design, implementation, and debugging processes. Figure 3-1 shows the TMS320C3x development environment supported by products from Texas Instruments.

Software tools available include an Assembler/Linker allowing programming in assembly language, an ANSI C Compiler so C source code may also be used, and a TMS320C3x Simulator to allow for source code debugging of programs. The TMS320C3x Simulator was used extensively during the coding portions of this thesis to test code prior to EPROM programming. All three of these software products are loaded on PCPWR 8 located in Bullard 114. Appendix A contains an explanation of how to access and run the Assembler, Linker, and Compiler by using DOS commands

and batch files. Further information concerning the specific features provided by these products is available in their respective reference manuals [Ref. 9 and Ref. 10] which are also located in Bullard 114. The TMS320C3x Simulator is described in Reference 11. Included in this manual are installation instructions and an operational tutorial that proved quite helpful.

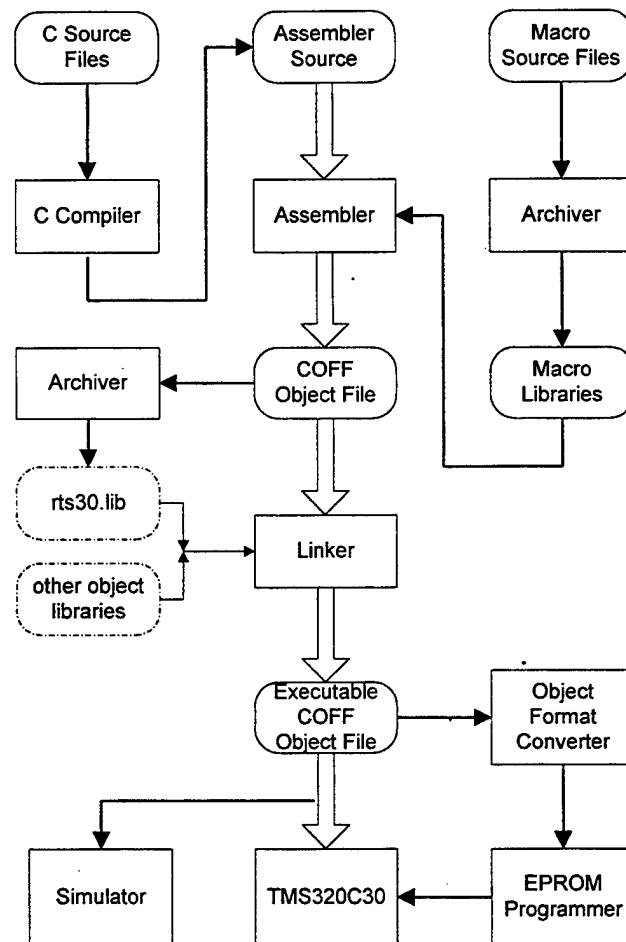


Figure 3-1 Development Support for the TMS320C3x [Ref. 8]

Other software products available and shown on Figure 3-1 are an Object Format Converter used to convert executable code into a format compatible with PLD programming and an EPROM Programmer to do the actual programming of PLDs.

Again, these products are loaded on PCPWR 8 in Bullard 114. Use of the Object Format Converter is addressed in Appendix A of this thesis and a complete description of how to use the EPROM Programmer can be found in Appendix C of Reference 3. Also available from Texas Instruments but not shown in Figure 3-1 is an XDS Emulator. This is a hardware device that allows full speed program execution of TMS320C3x programs. The Power Systems Lab at the Naval Postgraduate School (NPS) does not have this piece of hardware so it will not be discussed in detail here.

As has been shown, numerous resources are available from Texas Instruments to aid in the development of a TMS320C3x system. The C30 has a flexible instruction set with many addressing modes to allow flexibility in programming control algorithms. The architecture utilizes extensive paralleling to provide the speed needed for DSP control algorithms. The utilization of these characteristics as applied to buck chopper control is discussed in the next chapter.

IV. BUCK CHOPPER APPLICATIONS

A. INTRODUCTION

A major component of the DC ZEDS system is the Ships Service Converter Module. The SSCM is a feedback-controlled buck chopper used to step down a DC voltage to a lower level. Figure 4-1 shows a basic schematic of a buck chopper. A buck chopper uses an electronic switch to "chop" an input DC voltage and an LC-filter to eliminate the high-frequency components to produce a lower, average DC value.

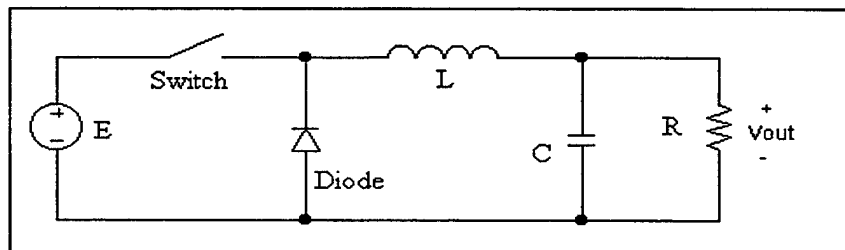


Figure 4-1 Buck Chopper Basic Schematic [Ref. 3]

In present Navy shipboard designs, the electronic switch is an Insulated Gate Bipolar Transistor (IGBT). Control signals applied to the gate of the electronic switch change the duty cycle of the switch and change the average DC voltage out. Figure 4-2 illustrates the affect that the duty cycle of the switch has on one average DC voltage out.

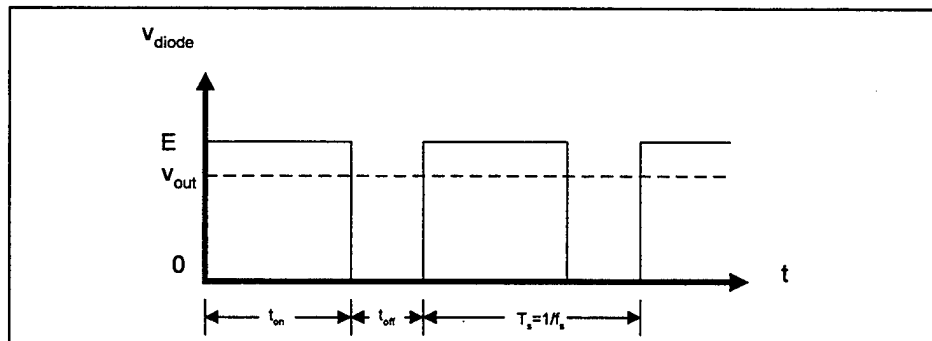


Figure 4-2 Average DC Voltage from a Buck Chopper [Ref. 3]

For continuous inductor current, the ideal steady-state relationship is given by

$$V_{out,ave} = D_{ss} * E \quad (4-1)$$

where

D_{ss} = steady-state duty cycle (t_{on}/T_s)

E = input voltage

Further details on buck chopper operation are listed in Reference 3.

Previous research developed a closed-loop control algorithm for the SSCM. The Universal Controller has enough I/O capability to simultaneously control 2 buck choppers. Reference 3 details single and dual buck chopper control. System integrators at NSWC required some modifications to the software control that involved incorporating additional features not currently available. This chapter addresses the added under-voltage, over-temperature and over-current protection algorithms and documents the changes to the assembly code provided in Reference 3. The next section describes how these changes were implemented. Another added feature was the Local/Remote switch. Section C of this chapter outlines the changes to the interrupt structure and the code required to allow operation of an L/R switch. Finally, the buck choppers must be capable of operating in parallel and sharing the load proportionately. Section D addresses this issue and describes the changes made to implement a Master/Slave type control algorithm.

B. PROTECTION CIRCUITS

1. 24 volt Control Power Low/Over-Temperature

Self protection requirements were established by personnel at NSWC for SSCM control operation. These requirements consisted of a 24 volt (control power) low shutdown, an over-temperature shutdown, and an over-current sense and shutdown. These changes needed to be incorporated in software. In the cases of 24 volt low and over-temperature, these changes merely consisted of reading a status words in from an external sense board that monitored these conditions and comparing the result to a word that corresponds to an admissible condition. The output from the sense board is connected to a general purpose I/O jack labeled connector JP-1 on the Universal Controller.

Bit 0	over-temperature slave
Bit 1	over-temperature master
Bit 2	under-voltage slave
Bit 3	under-voltage master

Table 4-1 Protection Circuit Bit Assignments

Table 4-1 shows the bit assignments used for the general purpose I/O connector on the Universal Controller. Memory address location 804500h (d_output) is associated with the general I/O port and was used to test for errors. Only the four (4) least significant bits needed to be checked, so the rest were masked out. Since the requirement was for any one of the fault conditions to shut down the bucks, only one compare was

needed. All four bits were tested at once. If any one of them indicated an error, the system was shut down. The following code was added to interrupt subroutine 0 (isr0):

```

LDI   @d_output,AR0    ;set pointer to Gen I/O
STI   *AR0,R0          ;R0=gen_I/O word
AND   000fH,R0         ;mask all but 4 lsbs
LDI   @cmd_ad,R4        ;Jump target if needed for shutdown
CMPI  0fH,R0            ;see if word is good
BNE   R4                ;Shuts down Bucks

```

Interrupt subroutine 0 was selected as the place to add the protection code because isr0 will run in both local or remote mode. Local and remote modes of operation are explained in a later section. The third protection circuit code, over-current sense and shutdown, was placed in isr0 for the same reason.

2. Over-Current Sense and Shutdown

The over-current sense and shutdown code was designed to monitor the average over-current and shut down the system whenever the average over-current exceeded 150% of rated current. The problem caused by over-current is that the heat that builds up in the solid-state switches has no time to dissipate. As the average over-current increases, the operating temperature of the device rises and eventually the component fails. But, when the heat has a chance to dissipate, indicated by when the average over-current decreases, the components have a chance to recover and system shutdown is not required. The average over-current was calculated by integrating the over-current over time. This was accomplished by encoding the following equation:

$$i_{\text{over-current_average}} = \int (i_{\text{out}} - 116) dt \quad (4-2)$$

where 116 represents 100% rated current in amps.

The actual encoding of the integral required several steps to accomplish. The integration was to be performed using the trapezoidal integration method as explained in Reference 3. Several lines of code and some new variables and constants had to be inserted to implement the integration digitally.

The first step in encoding the over-current sense and shut-down code was to create a variable named T/2 that is equal to one-half the switching period. This value is calculated in the cmd 10 subroutine of the buck chopper code provided in Reference 3 but was not saved for later use. The following line of code was inserted to accomplish this.

STF R0,*+AR3(tau_2) ; Store T/2

After saving T/2, several other variables and constants needed to be created.

Table 4-2 contains a listing of the added constants and variables and their initial values:

full	116.0
limit	58.0
io_m_116	0.0
trip_m	0.0
io_s_116	0.0
trip_s	0.0

Table 4-2 Added Variables for Over-Current Sense and Shut-Down

The constant 'full' represents 100% of rated current while 'limit' represents the maximum amount of average over-current allowed. The terms 'io_m_116' and 'io_s_116' are used to store the output over-current or $(i_{out} - 116.0)$ for the master and

slave buck choppers respectively. The Riemann sums or average over-currents for the master and the slave are stored in 'trip_m' and 'trip_s'.

The final step involved encoding the integral. Appendix B contains the entire code for the buck choppers with the over-current sense and shut-down portion inserted in isr0. Since the code for the master and the slave functions exactly the same, only the code for the master is outlined here. First, the value of 'full' is loaded into the microprocessor's general register seven (R7), then it is subtracted from the output current of the master. This creates the value of 'io_m_116[n]' which is stored for use in the next cycle as 'io_m_116[n-1]'. Next, the previous value of 'io_m_116' is added to the current value and multiplied by 'T/2'. The result represents the change in the average over-current for this small portion of time. This integral change is added to the previous total to produce the Riemann sum. If the sum is negative, the sum is reset to zero which assures a non-negative integral. The sum is stored for use during the next cycle and then compared to the limit. If the limit is exceeded, the subroutine branches to the cmd 0 subroutine which shuts down the system. If the limit is not exceeded, the subroutine exits this portion of code and resumes normal operation.

After coding the protection circuits, the next change provides for a Local/Remote (L/R) switch for system control and a front panel potentiometer adjustment for reference voltage in Local mode.

C. LOCAL/REMOTE SWITCH MODIFICATION

1. Specifications

System integrators at NSWEC requested that the buck choppers be equipped with some sort of front panel hard-wired controls to aid in troubleshooting and maintenance. A Local/Remote (L/R) switch that can select control of the buck choppers from either a front panel (Local mode) or from the host PC (Remote mode) was hard-wired in. The system needed to be able to be initialized and run in either local mode or remote mode and switched from one mode to the other during operation. When switched from mode to mode during operation, the system needs to shut down and restart in the new mode of operation because switching from one mode to the other would not be a "bumpless" transition. Two (2) voltage potentiometers were also connected to the front panel. They provided a course and fine adjustment for the reference voltage when the units were operating in Local mode. Software changes were required to enable operation of the front panel.

2. Application

In order to enable the front panel, the TMS320C30 had to be able to recognize and monitor the position of the L/R switch. Bit 5 of the general purpose I/O port, used previously by the protection circuits, was used to carry the L/R switch position. The reference voltage potentiometer signals were added together and sent as one input to the general purpose I/O port. This voltage was sent to an onboard A/D converter to create a voltage word usable by the Universal Controller. Once the TMS320C30 could access this information, changes to the software were made so it could process this information.

The normal program flow for the buck chopper control program was discussed in Chapter 2. As mentioned there, after initialization of the microprocessor, the program waits for an interrupt from the host PC. When the interrupt (interrupt 3) is received, the control values are loaded in from the PC, the Universal Controller initialization is completed and the program waits for phase interrupts to start controlling the system. When not processing an interrupt, the program is in a No Operation (NOP) loop waiting for the next interrupt. It continues this operation until it receives an interrupt from the PC to shut down the system. This defines remote operation. For local operation, the Universal Controller reads the reference voltage from the front panel potentiometers, uses this V_{ref} to calculate the changes to the duty cycle and therefore control the system. In order to start up the system in Local mode, a table of default control values had to be loaded into memory. Then, when starting up in Local mode, this table would be read instead of the control values from the host PC. As previously mentioned, switching from one mode to the other needed to cause a shut down of the system and a restart in the proper mode. Now that the modes of operation have been defined, the algorithm dictating how the Universal Controller monitors the L/R switch position and transitions between modes must be discussed.

It was determined that the interrupt structure of the control program would have to be changed. The Universal controller needed to check the L/R switch position on initial power-up and monitor its position throughout the entire operation of the program. It could no longer just wait for an interrupt from the PC. This created two (2) different start-up scenarios that had to be accounted for in the code, either start-up in Local mode or start-up in Remote mode. The Universal Controller needed to be able to

recognize a switch change from Local to Remote or vice versa. An internal timer interrupt was set up to check switch position. The previous switch position was saved to provide the ability to compare current position (sw[n]) with the previous position (sw[n-1]). This created four (4) possible run-time scenarios that also needed to be programmed into the code:

- 1) Switch previously in Local, stays in Local.
- 2) Switch previously in Local, switched to Remote.
- 3) Switch previously in Remote, stays in Remote.
- 4) Switch previously in Remote, switched to Local.

The interrupt structure and program flow were changed to cover all of the possible scenarios. Figure 4-3 shows a block diagram of the modified program flow allowing for L/R switch operation. The code is enclosed as a portion of Appendix B.

On initial power-up, the program initializes the microprocessor and then checks the position of the L/R switch. If in Remote, the program functions exactly as it did before. It waits for an interrupt 3 from the PC, continues the initialization process, and waits for phase interrupts. If the switch was in Local when checked, the program jumps to a routine that loads the default table into memory and then continues the program, waiting for interrupts. When an interrupt is received, if it is a phase interrupt (i.e. int0), the program runs the appropriate control algorithm encoded in the interrupt subroutine as before. Only if the interrupt is from the internal timer does the code change again.

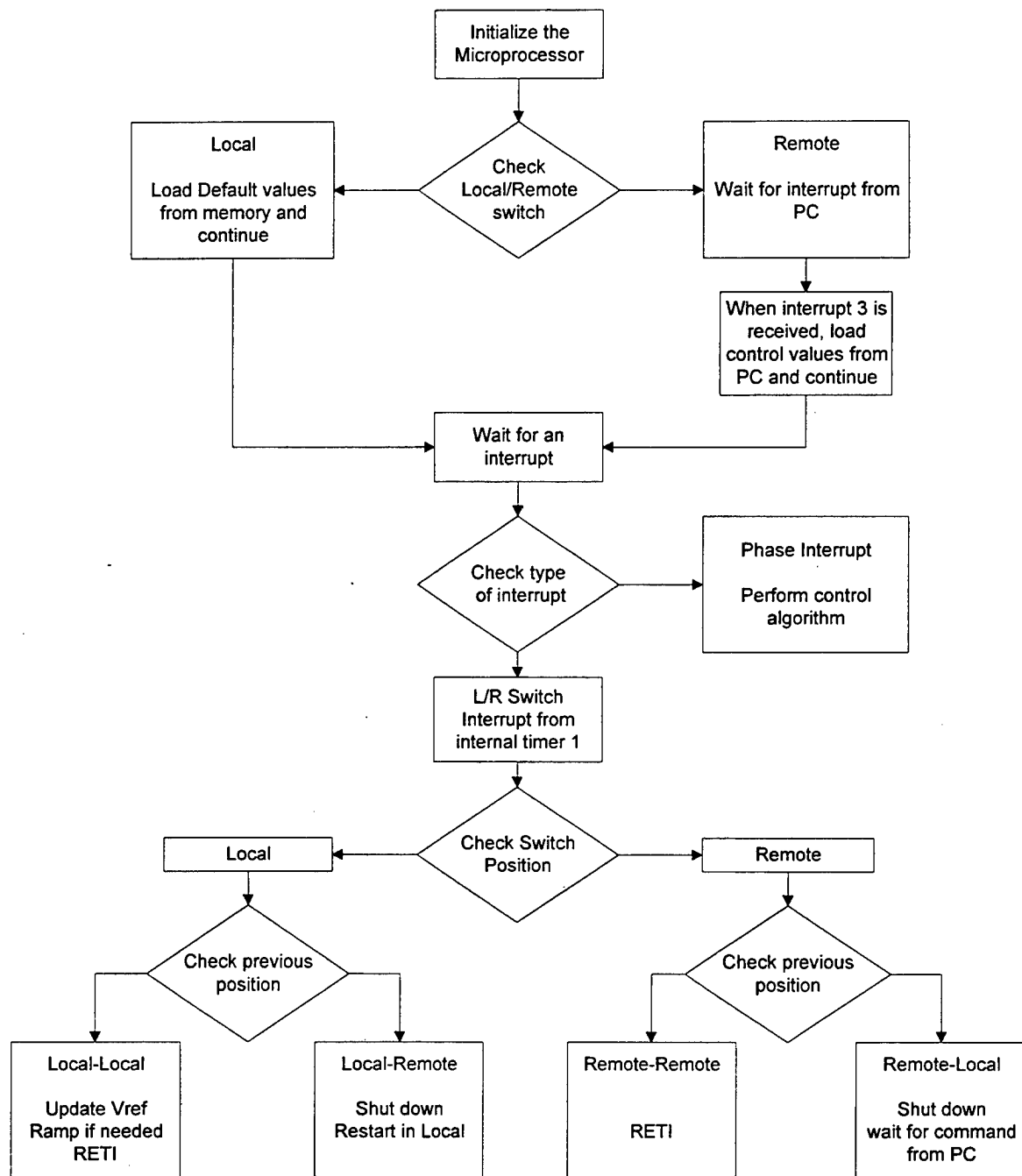


Figure 4-3 Program Flow with Local/Remote Switch

When an interrupt is received from the internal timer, the timer 1 subroutine takes over. First, it reads the previous switch position from memory then gets the current switch position. Zero (0) is used for remote and one (1) for Local mode. From

there it branches to the appropriate section based on current switch position. Once in the appropriate section of code, either Local or Remote, it compares the current switch position with the previous switch position to determine which of the four (4) run-time scenarios the system is in. The bottom of Figure 4-3 shows a brief synopsis of what is done for each of the four (4) cases. If in Local and previously in Local, the program updates Vref by reading in the front panel voltage, initiates a ramp-up function if Vref changed by more than 10 volts, stores the new Vref and returns from interrupt. If in Local and previously was in Remote, the program branches to the cmd 0 subroutine and shuts down the system. It then starts back up in Local. If in Remote and previously in Remote, the program merely returns from interrupt. Finally, if in Remote and previously in Local, the program branches to cmd 0 again to shut down the system and restarts waiting for an interrupt 3 from the PC to run in Remote mode. By using the TMS320C30's internal timer 1 to generate interrupts to check L/R switch position throughout operation of the system and then changing the interrupt structure to look for this interrupt, Local/Remote switch operation was encoded into the Universal Controller code. The only modification left to add for this thesis research involved encoding a Master/Slave algorithm for parallel operation of the buck choppers.

D. MASTER/SLAVE PARALLELING

1. Theory

The final requirement from NSWC was to investigate and develop a paralleling algorithm that did not require droop. NSWC personnel wanted to be able to connect two (2) 100 kW units together in parallel to create a single 200 kW unit. Reference 3

contains an algorithm for paralleling two (2) buck choppers based on decreasing the reference voltage specified by a unit as the unit output current increases (droop). The droop method worked but wasn't accurate enough for the 200 kW parallel application. Personnel at NSWC wanted current sharing accuracy greater than that attainable through the droop method and wanted to eliminate the droop or "sag" produced in the voltage as load increased. To correct this problem, a Master/Slave control algorithm was developed.

The basic premise behind the Master/Slave algorithm was to use a modified multi-loop feedback with a steady-state DC term, a Proportional plus Integral (PI) controller on the voltage, and a Proportional term on the current to calculate a duty cycle for the Master buck chopper. Input and output voltage can be established at one node because the buck choppers are in parallel.

$$V_{in,1} = V_{in,2} \quad (4-3)$$

$$V_{out,1} = V_{out,2} \quad (4-4)$$

However, the individual inductor currents must be summed and the individual output currents must be summed to establish base inductor and output currents.

$$i_L = i_{L1} + i_{L2} \quad (4-5)$$

$$i_{out} = i_{out1} + i_{out2} \quad (4-6)$$

The Slave buck chopper duty cycle then mimics the Master's. An Integral term is added to remove current error between the Master and the Slave, and a proportional term is added to maintain stability. Equations (4-7) and (4-8) show the control equations implemented using the Master/Slave scheme.

$$D_{master} = D_{ss} - h_v(V_{o1} - V_{ref}) - h_n \int (V_{o1} - V_{ref}) dt - h_i [(i_{L1} + i_{L2}) - (i_{o1} + i_{o2})] \quad (4-7)$$

$$D_{slave} = D_{master} + k \int (i_{o1} - i_{o2}) dt - k_p (i_{o1} - i_{o2}) \quad (4-8)$$

where,

D_{master} = Master Buck duty cycle D_{ss} = steady-state duty cycle

h_v = voltage gain V_{o1} = voltage out from Buck one

V_{ref} = reference voltage h_n = voltage integrator gain

h_i = current gain i_{L1} = Buck one inductor current

i_{L2} = Buck two inductor current i_{o1} = Buck one output current

i_{o2} = Buck two output current k_p = proportional gain

D_{slave} = Slave Buck duty cycle k = current integrator gain

2. Application

To implement the Master/Slave algorithm as efficiently as possible, much of the previous control code and interrupt structure was retained. Much of the code was written by Mr. Roger Cooley, an engineer for NSWC in Annapolis MD, with modifications made at Naval Postgraduate School to allow testing on the 20 kW units in the Power Systems Laboratory. Figure 4-4 documents the program flow for controlling two (2) buck choppers.[Ref. 3] The two bucks are controlled by the phase interrupts routed through PLD A and PLD B. The interrupts occur 180 degrees out of phase, or

every 25 msec, allowing the Universal Controller to monitor and control one (1) buck chopper at a time.

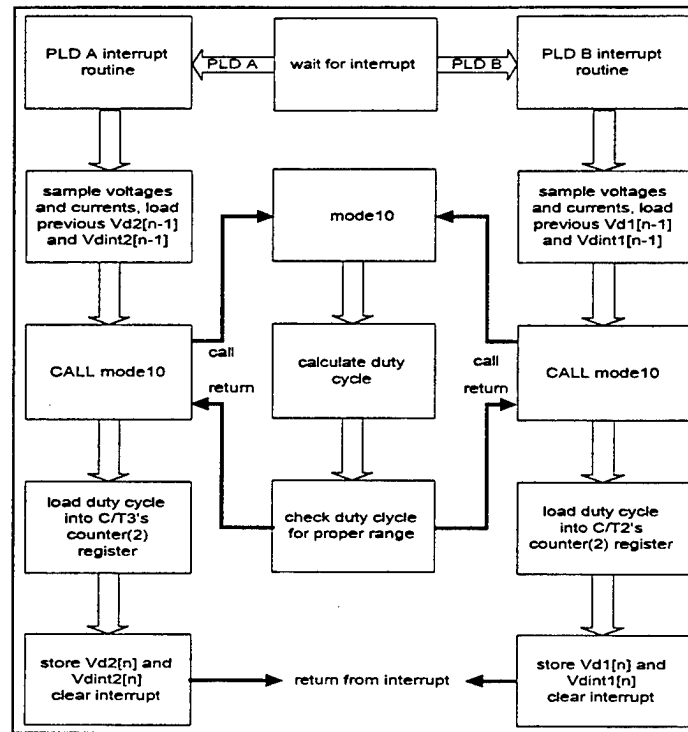


Figure 4-4 Program Flow for Dual Buck Chopper Operation [Ref. 3]

The actual designation of which buck would be the Master and which would be the Slave is purely arbitrary. Interrupt subroutine 0 was modified to control the Slave and interrupt subroutine 1 was modified for the Master. The actual encoding of the algorithm, the integration etc., was performed as it was in Reference 3. The program flow was changed as illustrated in by Figure 4-5.

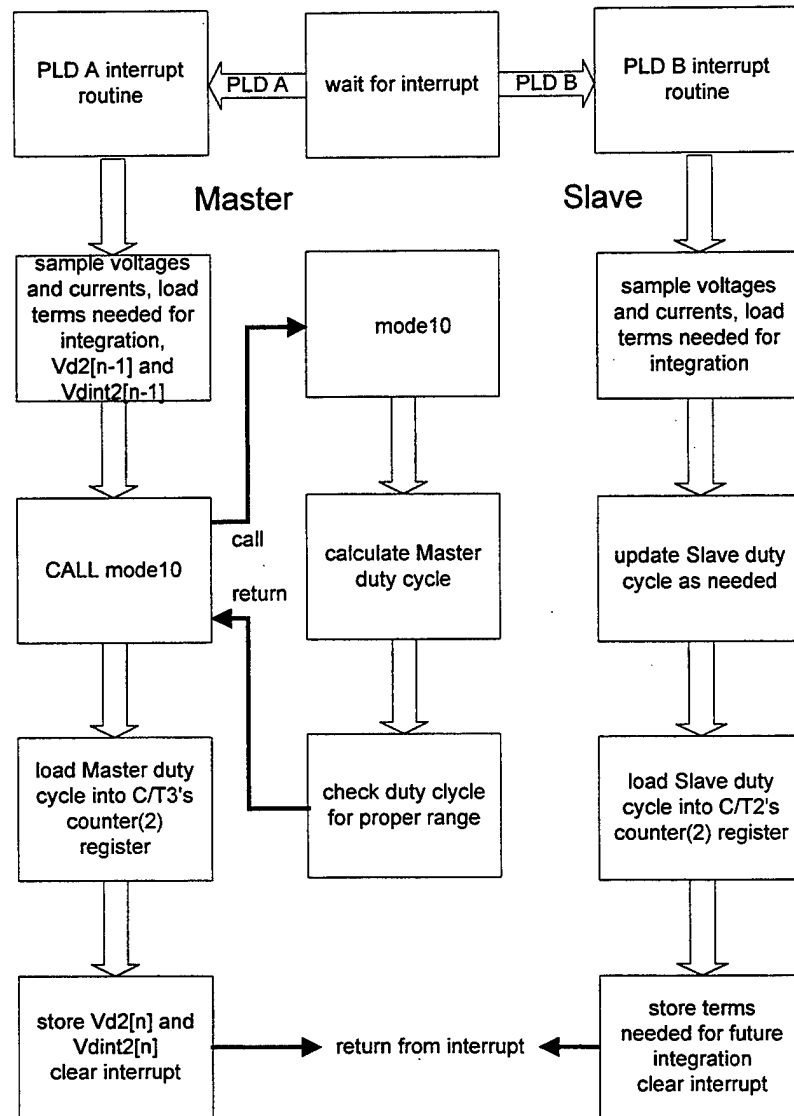


Figure 4-5 Master/Slave Modified Program Flow

3. Findings

The Master/Slave control algorithm performed satisfactorily on the 20 kW buck chopper units in the Power Systems Laboratory , Room 100A in Bullard Hall.

However, when tested on the 100 kW units at NSWC a problem was discovered. It was found that a circulating current due to the 180 degree phase shift was running from one buck to the other and was causing an error in the current readings measured during the

two (2) different phase interrupt subroutines. As the total load current increased, this differential current also increased and disrupted the proper current sharing desired by the two (2) units. Mr. Roger Cooley changed the program to read all values and perform all duty cycle calculations for both the Master and the Slave buck choppers during one phase interrupt. This “zero phase difference” design and code improved the performance of the Master/Slave algorithm. This code is also listed in Appendix B.

Thus far, all the programs for the control algorithms run by the Universal Controller have been written in assembly language. These programs are quite lengthy and complex. A high-level language, such as C, would increase readability of the code and reduce the time required for other researchers to understand the program operation. The next phase of this research dealt with investigating the use of C language programs for the Universal Controller.

V. C PROGRAMMING ISSUES

A. INTRODUCTION

There are two primary reasons that the C programming language was chosen to implement the control algorithms for the Universal Controller. First, the C language, being a high-level language, is more compact and more readable than assembly language. Closed-loop control algorithms with PI controllers use equations that involve relatively complex mathematical computations. The assembly code used to implement these equations is also complex and oftentimes difficult to decipher. Several lines of assembly code are required to do mathematical operations performed by a single line of C code. The second reason to use C was a matter of convenience.

An ANSI C compiler is supplied with the TMS320C30 microprocessor. This is a full-featured optimizing compiler that translates ANSI C programs into assembly language source code. [Ref. 9] The compiler allows for the interlacing of assembly language instructions into C code and also allows assembly modules to call C modules and vice versa. Implementing C code was going to be done in steps in order to provide a measure of testability on existing systems.

The current assembly code is quite extensive. To prevent 'reinventing the wheel,' the plan was to write only the control algorithm for the buck chopper in C code, leaving as much of the remaining code intact as possible. This would save a significant amount of time in coding because the majority of the existing program used for initializing the system could be used as an assembly module that called the control algorithm in C. The

new C code control algorithm could be compared to the already tested assembly language one (the buck chopper closed-loop control algorithm in Reference 3) to ensure operability and yet provide the flexibility desired for future modifications. Then, once the code was tested and working on the buck chopper system in the Power Systems Lab, the closed-loop algorithm for the ARCP inverter would be written in C. When completed, it would be inserted into the existing program that runs the inverter in open-loop mode. This approach would avoid the need to write complicated assembly code for the ARCP closed-loop algorithm and provide a readable, modifiable closed-loop algorithm. To write C modules that could 'talk' to the existing assembly code, several requirements had to be met.

B. C INTERFACE REQUIREMENTS

The TMS320C30 supports interlacing assembly language and C code with its onboard C compiler. It facilitates the writing of modules both in C and assembly language, compiling them both in a single step, and linking them together to form one executable object code. The two types of code will work together as long as some very specific rules are followed. These rules are outlined in Reference 9 (pages 4-10 through 4-25) and deal with variable naming and module calling conventions as well as proper register usage and parameter passing schemes used by the C compiler.

Assembly code modules can call C modules as long as the variables used by the C compiler are prefaced with an underscore (`_`) in the assembly code. For example, a variable used by C code called newcount needs to be listed as `_newcount` in the assembly language code and defined in the `.dss` section of the source code. This rule applies to all constants, variables, and module names called by the C code. This is because the C

compiler automatically prefixes all variable names called by a C function with an underscore (), so for the two codes to work together, this convention must be followed.

This naming convention was easily complied with by simply changing variable names in the assembly code. If any of the variables were overlooked, it became evident during the linking process when “unknown variable” errors surfaced. Further editing then allowed for error-free compiling and linking. The second requirement for interfacing code, the register usage/variable passing convention, proved more difficult to implement.

C. PROBLEMS WITH IMPLEMENTATION IN EXISTING CODE

In order for C code modules and assembly modules to communicate, strict register conventions must be followed [Ref. 9]. Table 5-1 summarizes the C compiler’s register use and preservation conventions.

Register	Use by Compiler	Preserved by Call
R0	Scalar Return Values	No
R1-R3	Integer and Floating Point Expressions	No
R4-R5	Integer Register Variables	Yes
R6-R7	Floating Point Register Variables	Yes
AR0-AR2	Pointer Expressions	No
AR3	Frame Pointer	Yes
AR4-AR7	Pointer Register Variables	Yes
IR0-IR1	Extended Frame Offsets	No
SP	Stack Pointer	Yes
RC, RS, RE	Block Copy	No

Table 5-1 Register Use and Preservation Conventions [Ref. 9]

This table shows the convention that must be followed when interfacing assembly language modules into C code. According to Reference 9, the called function is

responsible for preserving the contents of any used registers. In other words, when C code calls an assembly language function, the called assembly language function is responsible for saving and restoring any registers it modifies. Reference 9 further states that the C compiler must be free to modify registers as needed to accomplish program requirements which means that the compiler will choose which registers to save and restore based on Table 5-1. This issue is at the heart of the programming dilemma.

Instead of inserting assembly functions into C code, the previously stated programming plan intended to insert C code functions into an existing assembly language program. The problem this created is explained shortly. To further aggravate the situation, the register convention of the assembly code does NOT follow that stated by Reference 9. For example, C code uses AR3 as the frame pointer for the program code, but AR3 is used as a pointer to scratch pad memory and not the current working frame in the assembly code. Also, the assembly code uses general registers R0-R7 for all types of uses not just those specified by Table 5-1.

The problems created by inserting C code into assembly language programs that do not follow the stated register convention stem from the fact that the C programming language was designed to operate independently of system architecture. [Ref. 12] The compiler chooses which registers to assign values to, often based on some type of least cost algorithm. The compiler's algorithm decides which registers to save and restore at the time the program is compiled. The C language does not have provisions for mandating which registers are stored and which are not. This means that the calling assembly program would know which registers the C module would save and restore based on Table 5-1 but not which registers the C module would use or modify.

The first attempt at calling a C function from the assembly code failed because of this problem. Running the program on the TMS320C30 simulator showed an extensive amount of data that the assembly code was storing in registers for later use was being over written by the C code. To try and work around this register saving problem, the code was modified so that the calling assembly function saved all the registers before calling the C module and restored them after the return from the C module. This created a problem with the passing of variables from assembly to C and back again.

When all the registers were saved and restored each time a C module was called, the parameter passing ability from assembly to C was lost. In order for the C code to use any values with which to make any calculations, the values first had to be saved in memory locations accessible to the C module, upon which they could then be modified by the C code, and finally saved back in memory prior to returning to the assembly code. This complex scheme of saving all parameters in memory, then saving all registers on the stack prior to calling C code, then restoring all registers from the stack, and finally changing those values modified in memory proved more complex than just using the original assembly code. When this code was run on the simulator, data was still being lost due to a memory issue caused by the hardware.

Some of the values needed by the C code had to be read from the A/D converters on the Universal Controller. As stated earlier, these A/D converters have memory locations assigned to them and conversions are initiated by reading these locations. However, the memory map used by the TMS320C30 and therefore used by the compiler identified these locations as illegal memory locations and would not allow C memory

pointers to be assigned to them. In order to initiate conversions of the required input data, the microprocessor had to be forced to read these locations with assembly code.

After several weeks of trying to work around the register saving problem and the variable passing problem, it was decided to abandon the C code. The final program written in assembly language that called C modules ended up longer and more complex than the original assembly code. The final program still lost register information that prevented it from running properly when switching from assembly code to the C environment and back again. It was decided the assembly language program would have to be rewritten to allow the interface with C code modules. Extensive changes in the use of the registers by the assembly code would have to be made. The time required to do this proved too great for this thesis research.

Another option would be to write the entire program in C. This would require extensive research of the TMS320C30's online C run-time libraries and a thorough understanding of the C programming language. Then C modules could be written to initialize the microprocessor, provide the required interrupt structure, and initialize the Universal Controller counter/timers as needed for control algorithm implementation. This also proved too time consuming for this research effort. The decision was made to implement the closed-loop ARCP inverter algorithm in assembly language

VI. ARCP CONTROL

A. BASIC ARCP INVERTER OPERATION

The topology and operation of an Auxiliary Resonant Commutated Pole (ARCP) inverter is described in Reference 13. An operational unit was designed by individuals at the Applied Research Laboratory, Penn State University and delivered to the Power Systems Laboratory at NPS. It is designed to convert DC voltage into three-phase (3ϕ) AC using auxiliary semiconductor devices to implement soft switching. Below is a circuit block diagram of one phase of the ARCP inverter.

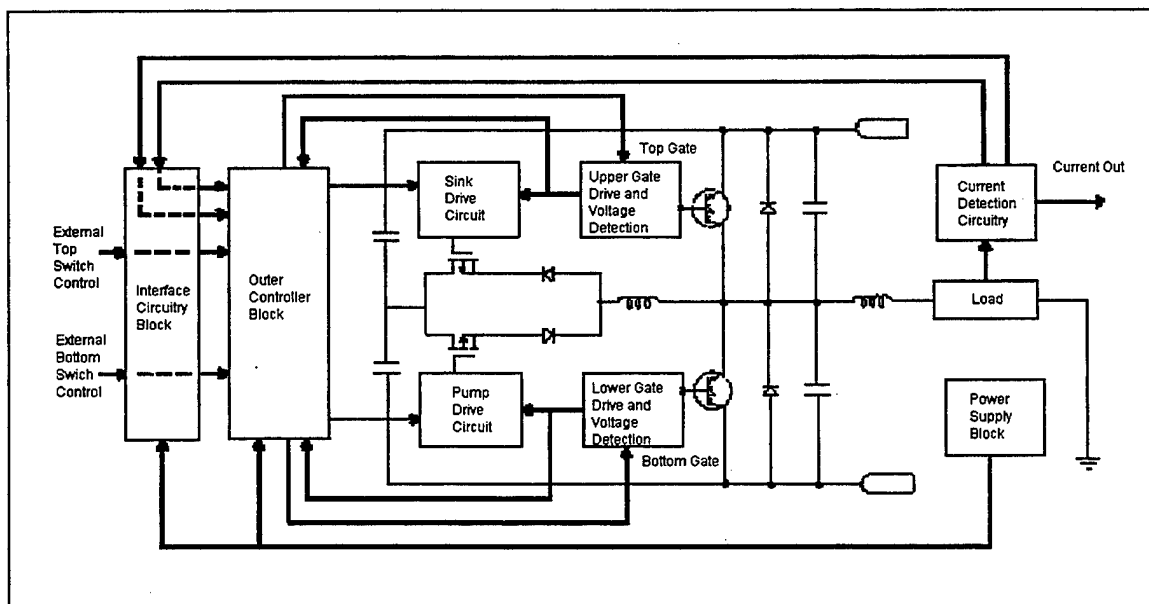


Figure 6-1 ARCP Inverter Circuit Block Diagram [Ref. 14]

The inverter has two primary switches and two auxiliary switches for each phase. The inverter operates by applying a changing control signal to the gates of these electronic switches. Each phase has two main drive circuits and two auxiliary switch drive circuits. Control signals from the Universal Controller act as inputs to the main drive circuits via

optical links. An inner control loop on each phase senses the direction of current flow and controls the operation of the auxiliary switches. [Ref. 13] During half the cycle, the pump drive circuit turns on the lower auxiliary switch and the inverter sources current. During the other half cycle, the sink pump circuit is operating the upper auxiliary switch and the inverter is a current sink. Onboard controls, which may be overridden, dictate the firing of the auxiliary devices. If the auxiliary device controls are not overridden, control of the unit can be accomplished by merely controlling the signal sent to the six (6) main electronic switches and will be discussed below.

B. OPEN-LOOP CONTROL

The open-loop control program provided by NSWC initializes the Universal Controller's counter/timers the same way it did for the buck choppers only with 120 degree displacement between phases. This phase difference is created in part by offsetting the phase interrupts that are used to calculate the control signal duty cycles. The code that actually performs this function can be found in the `init_swct` subroutine listed in Appendix C.

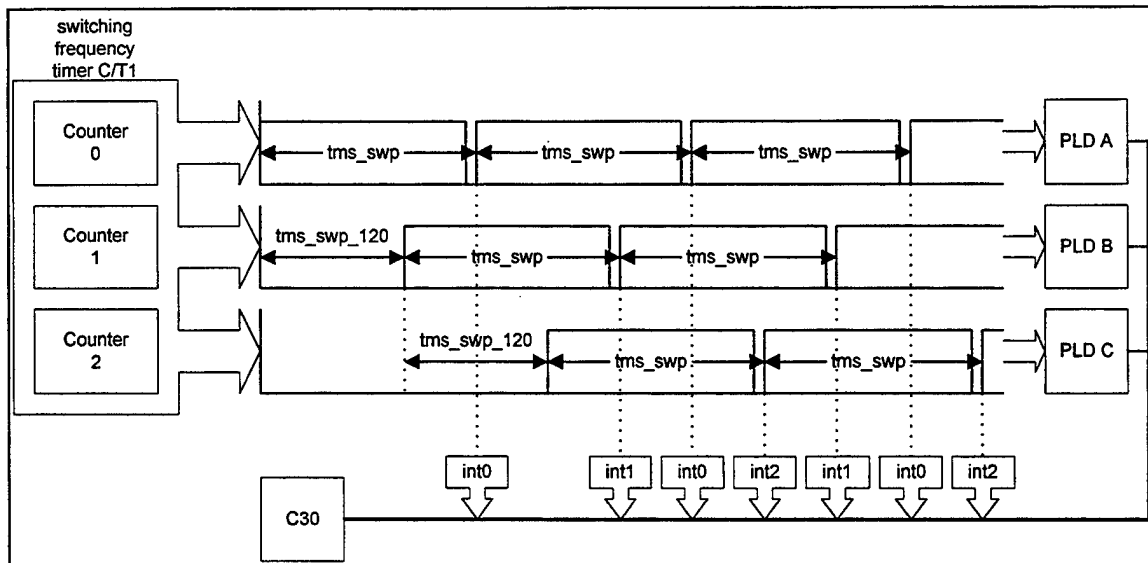


Figure 6-2 Three-Phase Interrupt Initialization [Ref. 3]

The switching period count is calculated from the switching frequency entered from the host PC and loaded into one of the switching frequency timers. This same count is then loaded into the next phase counter after being delayed by $2/3$ of the period and into the last counter after another similar delay. This ultimately produces a 120 degree difference between phase interrupts.

The duty cycle count for the primary switches is based on sine/triangle pulse-width-modulation (PWM). In PWM, a sine wave at the desired output frequency of the inverter is superimposed on a fixed-amplitude triangular wave at the desired switching frequency of the inverter. Each phase will have a similar sinusoidal control signal with the respective signals 120 degrees out of phase. When the sine wave is greater than the triangle waveform, the upper switch for the given inverter leg is gated. When the sine wave is less than the triangle waveform, the lower switch is gated. This creates a voltage across the lower switch as illustrated in Figure 6-3. The pattern basically may be viewed as a varying duty cycle applied to the devices in the inverter leg. The amplitude of the

sinusoidal control signal directly dictates the amplitude of the resulting phase voltage. Again, the switching period of the control signal is determined by the switching frequency entered from the host PC.

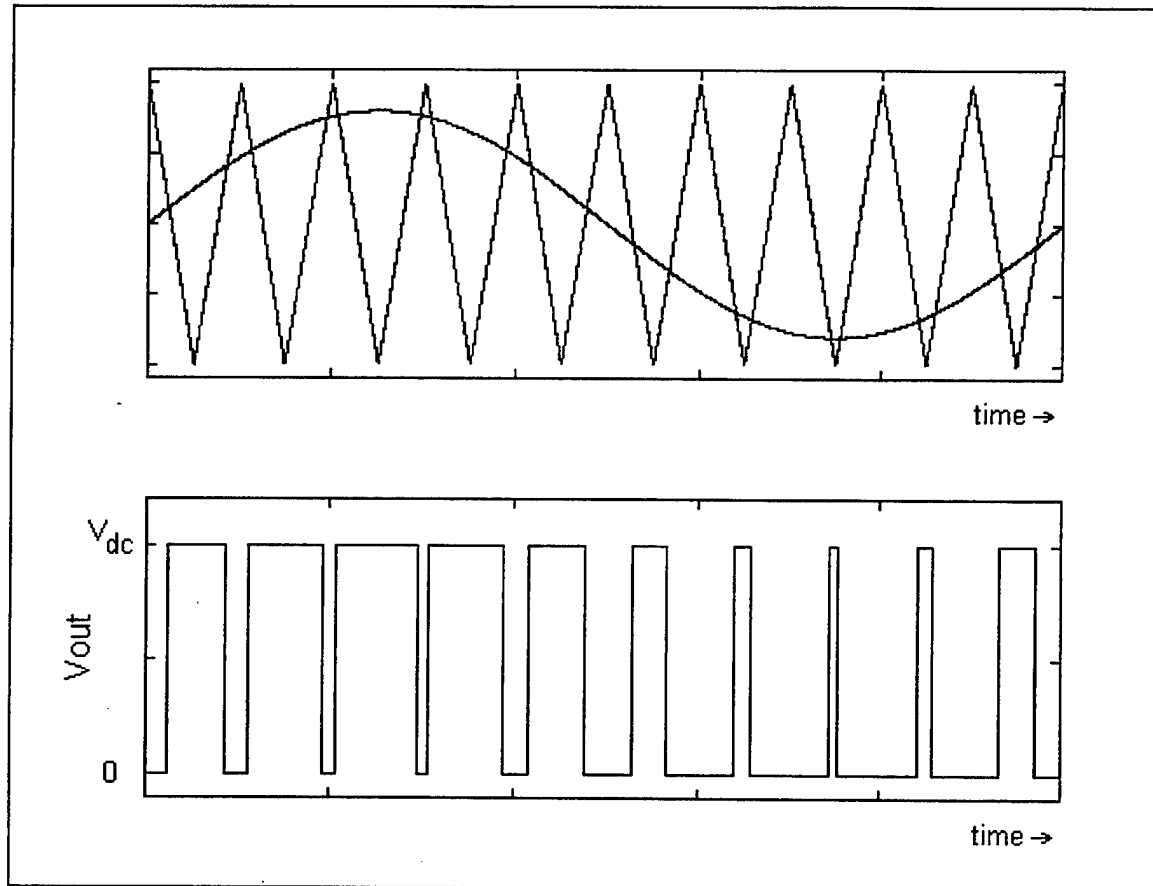


Figure 6-3 Sine/Triangle Pulse-Width Modulation [Ref. 2]

Figure 6-3 shows that as the amplitude of the sine wave approaches its maximum value, the duty cycle of the primary switch approaches its maximum. When the sine wave is at a zero value, the duty cycle is at 50% and decreases to a minimum at the maximum negative value of the sine wave. Equation (6-1) shows the assembly language formula that actually implements this type of modulation and calculates the duty count.

$$dutycount = tms_tb * R7 + tms_ta \quad (6-1)$$

where

$R7$ = value read from a sine-wave lookup table

tms_ta = sweep period/2

tms_tb = {sweep period - 2*(dead time)}/2.

Using ta and tb in this manner has the affect of shifting the sine wave output up so the minimum is at or near zero. This is required for implementation because it is impossible to load a negative count or have a negative duty cycle. Dead time is used to ensure proper switch operation. Dead time usually refers to the time between turning the top switch off and the lower switch on. Both switches conducting at the same time would cause catastrophic failure of the unit. The ARCP inverter used in this research has circuitry onboard that controls this situation. The dead time referred to in tms_tb is used to prevent the duty cycle limits of 5% to 95% from being exceeded. Sine theta is determined by the use of a look-up table loaded into scratch pad memory and by using the circular addressing mode of the microprocessor. A detailed explanation of circular addressing can be found in Reference 8. Below is an example of an instruction that uses circular addressing.

LDF...*AR7++(IR1)%,R7

Basically, an auxiliary register is used as a pointer to the look-up table and an index register is used as a step size index. The step size tells the pointer how far to step or index after reading the current value of the table. In this case, the value of the sine table that pointer AR7 is pointing to is loaded into R7 then the pointer is indexed by the

amount in IR1. If the end of the table is reached, the pointer circles around and starts at the beginning of the table again.

The three phases are kept 120 degrees apart by using three (3) different pointers, one for each phase, and a double circular addressing scheme for the Phase B and C pointers that keeps them tied 120 degrees out of phase with the Phase A pointer. Below is the code for the double circular addressing scheme used for one of the Phase B or C interrupts. The only difference between the Phase B and Phase C scheme is the value of IR1. For Phase B it is equivalent to 120 degree displacement and for Phase C it provides a 240 degree displacement.

```
LDI    AR7,AR6      ;  
LDF    *AR6++(IR1)%,R7;  
LDF    *AR6++(IR1)%,R7;
```

Recall from above that AR7 is the Phase A pointer to the sine table. The first line of code copies this pointer to AR6 so that when indexing is done, AR7 is not changed. AR7 should only increment during the Phase A interrupt. Line one of the code prevents AR7 from indexing during the Phase B or Phase C interrupt. The next line is a regular circular addressing instruction. It loads R7 with the value pointed to by AR6 which in this case equals the current Phase A sine value. AR6 is then incremented the appropriate 120 or 240 degrees. The circular addressing instruction is then used again to load the desired sine table value with the proper phase displacement into R7 for use in Equation 6-1. This double circular scheme is what allows the duty count for each of the three phases to be calculated during different interrupt subroutines in the program yet remain exactly 120 degrees apart.

Finally, using Equation 6-1, the new duty count is calculated for each phase and loaded in the corresponding counter/timers. This produces the required modulation signal which is then exported to the primary switches providing open-loop operation. Further details about open-loop operation of the ARCP can be found in Chapter 5 of Reference 3. The next step was to close the loop on the control algorithm.

C. CLOSED-LOOP CONTROL

1. Theory

Closed-loop control, as mentioned earlier, is a method of controlling a system that uses a portion of the output fed back to modify system operation. This is done to reduce or eliminate transients and steady state inaccuracies caused by a changing load or changing inputs. Closed-loop control can be accomplished by using either a current control mode or a voltage control mode. The current control mode was selected in this case because the ARCP inverter already has sensors in place that provide scaled measurements of the system currents. The current control mode allows the inherent limiting of the current flowing through the semiconductor switches.

Control signals for the ARCP inverter can be established by regulating either stationary reference frame quantities or synchronous reference frame quantities. As discussed in Reference 2, using commanded quantities in the synchronous reference frame are preferred over the stationary reference frame because the steady-state commanded values are DC levels in the synchronous reference frame. In other words, when operating in the synchronous reference frame and in a steady state (no perturbations present), the error term produced by a PI controller will be zero. The annotation for the control algorithm is shown in Table 6-1.

Superscript 's'	stationary reference frame quantities
Superscript 'e'	synchronous reference frame quantities
Subscripts 'abc'	actual phase quantities
Subscripts 'qd'	transformed quantities
Superscript '*'	commanded or reference quantity

Table 6-1 Closed-Loop Control Algorithm Annotation

The first step of closed-loop control was to sample two (2) phase currents, i_a and i_b . This will exploit the fact that for a three-wire wye-connected AC load, the sum of the three instantaneous phase currents is zero. The two (2) phase currents were then transformed into the synchronous reference frame. To make this transformation easier to follow, it was performed in steps. First, the measured quantities were transformed into the stationary reference frame using the following diffeomorphic relationship:

$$\begin{bmatrix} i_q^s[n] \\ i_d^s[n] \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -\frac{\sqrt{3}}{3} & -\frac{2\sqrt{3}}{3} \end{bmatrix} \begin{bmatrix} i_a[n] \\ i_b[n] \end{bmatrix} \quad (6-2)$$

Once in the stationary reference frame, the following transformation is applied, placing the quantities into the synchronous reference frame.

$$\begin{bmatrix} i_q^e[n] \\ i_d^e[n] \end{bmatrix} = \begin{bmatrix} \cos(\theta_e[n]) & -\sin(\theta_e[n]) \\ \sin(\theta_e[n]) & \cos(\theta_e[n]) \end{bmatrix} \begin{bmatrix} i_q^s[n] \\ i_d^s[n] \end{bmatrix} \quad (6-3)$$

Here, θ_e is the electrical angle of the measured quantities. The values of $\sin(\theta_e)$ and $\cos(\theta_e)$ are found using the sine look-up table described in the previous section. As mentioned then, a pointer was set for the sine and double circular addressing was used

for the cosine as before. Once i_q^e and i_d^e are calculated, they are compared to commanded values entered from the host PC, i_q^{e*} and i_d^{e*} . This produced i_{qq} and i_{dd} as shown by Equation (6-4) and (6-5).

$$i_{q,q} = i_q^{e*}[n] - i_q^e[n] \quad (6-4)$$

$$i_{d,d} = i_d^{e*}[n] - i_d^e[n] \quad (6-5)$$

The values $i_{q,q}$ and $i_{d,d}$ are next applied to a PI controller to calculate control voltages, $V_{q,PI}^e$ and $V_{d,PI}^e$. The PI controller equations are given as follows:

$$V_{q,PI}^e[n] = K_{pq}(i_{qq}) + K_{iq} \int i_{qq} dt \quad (6-6)$$

$$V_{d,PI}^e[n] = K_{pd}(i_{dd}) + K_{id} \int i_{dd} dt \quad (6-7)$$

The control voltages are now inverse transformed to the stationary reference frame using the following:

$$\begin{bmatrix} V_{q,PI}^s[n] \\ V_{d,PI}^s[n] \end{bmatrix} = \begin{bmatrix} \cos(\theta_e[n]) & \sin(\theta_e[n]) \\ -\sin(\theta_e[n]) & \cos(\theta_e[n]) \end{bmatrix} \begin{bmatrix} V_{q,PI}^e[n] \\ V_{d,PI}^e[n] \end{bmatrix} \quad (6-8)$$

These stationary reference frame control voltages, $V_{q,PI}^s$ and $V_{d,PI}^s$, are then converted to the three different phase control voltages or 'abc' quantities.

$$V_{a,PI} = V_{q,PI}^s \quad (6-9)$$

$$V_{b,PI} = -\frac{1}{2} V_{q,PI}^s - \frac{\sqrt{3}}{2} V_{d,PI}^s \quad (6-10)$$

$$V_{c,PI} = -V_{a,PI} - V_{b,PI} \quad (6-11)$$

Finally, the phase control voltages are used to calculate the new duty counts needed to produce the desired three-phase (3 ϕ) AC voltages. The equations for

calculating duty count are listed below. This is the same method used in the open-loop operation (Equation 6-1) except the phase control voltages are used in place of t_b . The new duty counts are then loaded into the appropriate counter/timers to produce the control signals sent to the ARCP inverter.

$$dutycount_a = V_{a,PI} * \sin \theta + tms_ta \quad (6-12)$$

$$dutycount_b = V_{b,PI} * \sin \theta + tms_ta \quad (6-13)$$

$$dutycount_c = V_{c,PI} * \sin \theta + tms_ta \quad (6-14)$$

2. Application

The code used to implement the closed-loop control algorithm is enclosed as interrupt subroutine 0 of Appendix C. Many of the techniques used in previous applications were used in this code. The same trapezoidal integration scheme was used for calculating the integrals needed by the PI controller and circular addressing with a sine wave look-up table was used for angle calculations. One important difference to note is that only one interrupt was needed for closed-loop control instead of the three used for open-loop operation. This involved making a slight change to the interrupt structure. Interrupt Subroutines 1 and 2 were completely eliminated so as to not disrupt operation of the now longer Interrupt Subroutine 0. The 120 degree difference between phases is maintained by sampling the currents of two phases and then using those two samples to calculate what all three duty counts should be when separated by the proper phase difference. The ARCP code, in its entirety, is enclosed as Appendix C.

VII. CONCLUSIONS

A. SUMMARY OF RESEARCH WORK

The PEBB Universal Controller is a digital controller designed to handle the extensive I/O requirements needed to implement closed-loop control of buck chopper converters and ARCP inverters. It provides great flexibility and is a valuable tool in the Navy's efforts to implement a DC ZEDS scheme. The focus of this research was to expand the operational capabilities of the buck chopper converter control algorithm and to implement closed-loop control of the ARCP inverter.

In Chapter II, the PEBB Universal Controller operation and architecture were investigated. The exceptional I/O capability of the Universal Controller was discussed as was the architecture of the CPU board. Basic program operation was outlined in order to set the stage for subsequent modifications. The Texas Instrument TMS320C30 microprocessor was discussed in Chapter III. The architecture, powerful instruction set, and paralleling hardware give the microprocessor exceptional speed and the ability to handle the tasking of the Universal Controller. Some of this tasking was covered in Chapter IV. NSWC personnel specified additional software features that had to be incorporated in the assembly language program governing the operation of the buck choppers. In Chapter IV the over-current, under-voltage, and over-temperature protection schemes were introduced and added to the control code. This chapter also covered modifications needed to provide for Local/Remote switch operation as well as encoding a Master/Slave paralleling algorithm. Improving the readability of the control

algorithms was addressed in Chapter V together with assessing the possibility of using C code. The several problems that were associated with trying to inject C modules into the assembly code were discussed. Finally, Chapter VI covered the control of the ARCP inverter. Basic open-loop operation was discussed and one closed-loop control algorithm was described and encoded.

B. NOTABLE CONCLUSIONS

The Universal Controller is very flexible. Modifications to the control algorithms can be made but the assembly language program is quite lengthy and complex. With no user's manual and little documentation available, a significant amount of time is required to understand the code well enough to make changes to the control algorithms. C code algorithms would be easier to read and modify but extensive rework of the existing program would be required to allow the interlacing of C code modules with the assembly code. Closed-loop control of the ARCP is achievable with the Universal Controller allowing fast, accurate response to system perturbations.

C. RECOMMENDATIONS FOR FUTURE WORK

Further work in the area of paralleling buck choppers is needed. The problems associated with the original 'droop method' and the differential cross current in the Master/Slave algorithm leave the door open for the development of an algorithm that would allow control of multiple bucks in parallel yet be able to ensure proper load sharing. Possible areas of investigation include current share wire or frequency injection controlling.

Research needs to be done to refine the closed-loop ARCP inverter algorithm including the possible move to a more specialized control card or even a commercial

control card that may not be as I/O capable or flexible as the Universal Controller but may allow different programming schemes. Another possibility would be to develop a new control program scheme that is not interrupt driven and that can be implemented in C, C++, visual C++ or some other high-level language or even possibly completely rewriting the existing program in C.

The U.S. Navy is looking for ways to save money and at the same time upgrade operational capabilities. DC ZEDS research is one way of meeting these goals. The flexibility of the Universal Controller makes it a key component in the DC ZEDS system and one worthy of further research and development.

APPENDIX A. SOFTWARE ACCESS AND DOS COMMANDS

A. PROGRAM DEVELOPMENT SOFTWARE TOOLS

In addition to the assembly programs discussed in this thesis, several software programs are required for implementation of this research. Appendix C of Reference 3 contains a detailed description of several support programs and their use. The programs are stored on PCPWR7 in Bullard Hall Room 114 and are installed in various computers throughout the Power Systems Laboratory. Reference 3 outlines how to install and use the Host PC software and how to install or "burn" code on the EPROMS used by the Universal Controller. The Host PC software is written in C++ and designed for use in a Windows 3.1 operating environment. The software required to install code on the Universal Controller's EPROMs is loaded on PCPWR8, also located in Bullard Hall Room 114. This is a DOS-based system. Batch files control much of the software on this system; therefore, a working knowledge of DOS commands and batch files is required. This Appendix assumes the reader has a basic knowledge of computers and the use of Windows but that the reader has had little exposure to DOS.

B. DOS COMMANDS

DOS is the precursor to Windows as an operating system for computers. Unlike Windows with its graphical user interface, DOS relies on a series of commands entered at command prompt to govern operation. Below is an example of a DOS command prompt:

```
C:\>
```

The “C:” refers to which drive is the current working drive and the backslash (\) with no other directories after it indicates that the computer is working in the root directory. The “C:” drive is usually the computer’s hard drive or primary memory storage. Commands, usually letters or short words typed from a keyboard, directly after the command prompt direct the computer’s operations. For the purposes of this Appendix, the command prompt is shown with each command discussed. Commands entered by the user are in **boldface**, and are directly related to using the support programs required for this thesis.

After applying power to the computer (PCPWR8), the system “boots-up” and the command prompt appears on the screen. To obtain a listing of directories present on the C: drive, type **dir** and press enter.

```
C:\> dir
```

The screen will scroll through a listing of all files and sub-directories located in the root directory. On PCPWR8, the list is too long to fit on the screen. To view the list one page at a time, type:

```
C:\> dir /p
```

This will display the entire listing one page at a time. The files consist of a filename followed by an extension. For example, on the file `npsbuck.asm`, `npsbuck` is the filename and `.asm` is the extension. DOS filenames are limited to eight (8) characters and can be either upper or lower case (DOS is not case sensitive). Extensions are used to define the file type. Table A-1 lists the DOS extensions most used in this thesis work and the meaning of each extension.

Extension	Meaning
.asm	Assembly language file
.c	C language file
.obj	object file produced by the assembler
.out	output file produced by converting an object file into the format needed to program EPROMS
.cmd	command file used to link format information to the assembly code for inclusion in the object code
.bat	batch file of executable instructions
< dir >	sub-directory

Table A-1 DOS Extensions

Assembly language code that is to be assembled must have the filename extension .asm and C language code that is to be compiled must have the filename extension .c. This code may be written using any text editor the user is familiar with as long as it is saved with the proper extension. As described in Reference 3, the C compiler and the Assembly language assembler are loaded on PCPWR8 in the DSPTOOLS sub-directory on drive C. To switch computer operation to this sub-directory type:

```
C:\>cd dsptools
```

The command "cd" stands for change directory and the command prompt will change to indicate the new working directory.

```
C:\DSPTOOLS>
```

The user is now ready to run the batch file, npsbuck.bat, as described in Reference 3.

C. BATCH FILES

Batch files are files that consist of a series of executable DOS commands. To run a batch file, the user simply needs to type the name of the file. For example, to run the batch file that assembles the file npsbuck.asm, the user must type:

```
C:\DSPTOOLS>npsbuck
```

As outlined in Reference 3 Appendix C, the input filename is required to be npsbuck.asm. A listing of the actual batch file explains why this is the case. To view the batch file, the following command is entered at the command prompt:

```
C:\DSPTOOLS>type npsbuck.bat
```

The batch file will then be displayed on the screen

```
asm30 npsbuck.asm -s -l -q  
lnk30 npsbuck.obj npsbuck.cmd  
hex30 -I npsbuck
```

The first command assembles the file named npsbuck.asm. The letters after the filename are different options available with the assembler and are defined in Reference 8. For instance, the -l tells the assembler to create a listing file and the -q suppresses the banner and all progress information during assembly [Ref. 8]. The second command links the object file produced by the assembler with the command file named npsbuck.cmd. The final command converts the object code into the proper hex format necessary to program an EPROM. The specifics of each instruction and their various options are described in Reference 8.

Batch files are very versatile. They can be written to perform a myriad of functions. To edit a batch file, use the DOS command `edit`. The following is typed at the command prompt

```
C:\DSPTOOLS>edit npsbuck.bat
```

An onscreen editor will appear with the contents of the batch file displayed. The file may now be changed as desired. The arrow keys on the keyboard will move the cursor. Any command that is present may be changed or any executable DOS command can be added to or deleted from the file. For instance, `npsbuck.asm`, located in the first line of the batch file, could be changed to a different filename thus allowing different assembly language files to be assembled.

During the course of this research, it became helpful to write program code on a different computer, save the file on a floppy disk, and assemble the code as discussed in Reference 3. In DOS, the "a:" drive is usually the floppy disk drive. To access this drive, the command "a:" is used. This command can be used in conjunction with other commands to add flexibility to the batch file. For instance, if line one of `npsbuck.bat` is changed to read:

```
asm30 a:npsbuck.asm -c -l -q
```

The assembler will now assemble the program `npsbuck.asm` that is located on the floppy disk in the a: drive. Another example using the "a:" command is to add the line:

```
copy npsbuck.out a:
```

to the bottom of the batch file `npsbuck.bat`. This would copy the output file created by the assembler/linker to the floppy disk located in drive a:.

The "a:" command can be used outside of a batch file as well.

```
C:\DSPTOOLS>copy a:npsbuck.asm
```

This command will copy the file named npsbuck.asm from the floppy disk in drive a: to the current working sub-directory, dsptools.

Knowledge of DOS is invaluable for performing research involving the Universal Controller. The most helpful commands have been discussed in this Appendix. A more thorough explanation of the workings of DOS and DOS commands is contained in Reference 17, an MSDOS 6 User's Guide.

APPENDIX B. BUCK CHOPPER CONTROL CODE

```

*****
*
*       NPS POWER LAB
*       TMS320C30 SSCM CONTROL CODE
*       BY RON HANSON
*       MODIFIED FOR PARALLEL OPERATION
*       BY BOB ASHTON  NPS (Theoretical)
*       ROGER COOLEY  NSWG (Coding)
*
*       Single Interrupt, Zero phase difference
*
*       OVER-CURRENT, UNDER-VOLTAGE, OVER-TEMP
*       LOCAL/REMOTE SWITCH OPERATION
*       BY DAVID FLOODEEN
*
*****
*
*       .title  "BUCK"
*       .global init
*
;----- EPROM Config
*       .global reset
*       .global int0,int1,int2,int3
*       .global tint0
*       .global isr0,isr1,isr2,isr3
*       .global time0
*
;----- END EPROM
*       .global SIN,FPINV,FDIV,divi
*
;-----
A2Dfltr .macro SRC, MSB

! Takes two 12bit values in:
!           b0..b11 (LSB) and
!           b16..27 (MSB)
! of the SRC and converts the two values into 32bit integer format
! Storing the LSB integer in the SRC register and
! Storing the MSB integer in the MSB register
!-----
! The arguments should be Registers
!-----
        LDI SRC, MSB
        LSH 04H, MSB
        ASH -14H, MSB
        FLOAT MSB
        LSH 14H, SRC
        ASH -14H, SRC
        FLOAT SRC
        .endm
;-----
*

```

```

        .text
init:  NOP      ;
        LDI 0,DP      ;==(EPROM)== Point the DP register to page 0
        ;==(Boot)LDI 08H,DP      ;==(Boot)== Init DP register
        LDI 00H,ST      ; Clear and enable cache, and disable OVM (1800h)
        LDI 0000H,IE      ; Clear all interrupts
        LDI @ctrl,AR0      ; Load peripheral bus memory-mapped reg
        LDI @xbus,R0      ;
        STI R0,*+AR0(60H) ; Init expansion bus control reg
        LDI @pbus,R0      ;
        STI R0,*+AR0(64H) ; Init primary bus control reg
        LDI @stck,SP      ; Initialize the stack pointer
        CALL init_ct      ; Init counter/timer
        CALL init_values  ; load default value table (L/R_sw)
        LDI @d_output,AR0 ;
        LDI 00FFH,R0      ;
        STI R0,*AR0      ;
        LDI @ct_port,AR0  ; Pointer for counter/timer control register
        LDI @reset_out,R0 ;
        STI R0,*AR0      ; Disable all output
        LDI @ctrl,AR0      ;
*
*
*
        LDI @blk1,AR3      ; Scratch pad memory area
        LDI @dp_mem,AR4      ; Top of dual port memory
        LDI @blk0,AR5      ; Scratch pad memory area
        LDI @sram,AR7      ; Top of the look up table
        LDI @dp_cint,IR0    ; Clear dual port memory interrupt
        LDI *+AR4(IR0),R0    ;
;---- LDI 000H,R0      ; Clear sram memory
;---- RPTS 2047      ;
;---- STI R0,*AR4++(1) ;
        LDI @dp_mem,AR4      ; Top of dual port memory
        LDI 0000H,IF      ; Clear all flags
        LDI 0200H,IE      ; Enable interrupt 9 (internal timer 1) (L/R_sw)
        OR 0200H,ST      ; Global interrupt enable
*
*
begin: NOP
        NOP
        NOP
        NOP
        BR begin
*
*
* Initialize counter/timer
*
init_ct: LDI @ct_port,AR0  ; Pointer for counter/timer control register
        LDI 00ffH,R0      ;
        STI R0,*AR0      ; Disable all counter/timer output
        LDI @ct_swfreg,AR0 ; Pointer for switching frequency timer 1
        LDI 0034H,R0      ; Mode 2 (rate generator), 00110100B
        STI R0,*+AR0(3)    ;

```

```

LDI 0074H,R0 ; 01110100B
STI R0,*+AR0(3) ;
LDI 00b4H,R0 ; 10110100B
STI R0,*+AR0(3) ;
LDI @ct_phasea,AR0 ; Pointer for phase a counter
LDI 0012H,R0 ; Mode 1 (hardware retriggeable one-shoot),00010010B
STI R0,*+AR0(3) ;
LDI 0052H,R0 ; Mode 1, R/W LSB, 01010010B
STI R0,*+AR0(3) ;
LDI 00b2H,R0 ; Mode 1, R/W LSB & MSB, 10110010B
STI R0,*+AR0(3) ;
LDI @ct_phaseb,AR0 ; Pointer for phase b counter
LDI 0012H,R0 ; Mode 1 (hardware retriggeable), R/W LSB, 00010010B
STI R0,*+AR0(3) ;
LDI 0052H,R0 ; Mode 1, R/W LSB, 01010010B
STI R0,*+AR0(3) ;
LDI 00b2H,R0 ; Mode 1, R/W LSB & MSB, 10110010B
STI R0,*+AR0(3) ;
LDI @ct_phasec,AR0 ; Pointer for phase c counter
LDI 0012H,R0 ; Mode 1 (hardware retriggeable), R/W LSB, 00010010B
STI R0,*+AR0(3) ;
LDI 0052H,R0 ; Mode 1, R/W LSB, 01010010B
STI R0,*+AR0(3) ;
LDI 00b2H,R0 ; Mode 1, R/W LSB & MSB, 10110010B
STI R0,*+AR0(3) ;

```

```

LDI @ctrl,AR0 ; Pointer for counter/timer control register (L/R_sw)
LDI @tim1prd,R0 ; load internal timer1 period (L/R_sw)
STI R0,*+AR0(38H) ; (L/R_sw)
LDI @tim1ctl,R0 ; init timer1 (L/R_sw)
STI R0,*+AR0(30H) ; (L/R_sw)

```

RETS

*

init_values: LDI @oaci,*+AR3(tms_oaci) ;load all front panel values for L/R_sw

```

LDI @acv,*+AR3(tms_acv) ;(L/R_sw)
LDI @bdly,*+AR3(tms_bdly) ;(L/R_sw)
LDI @btime,*+AR3(tms_btime) ;(L/R_sw)
LDI @dci,*+AR3(tms_dci) ;(L/R_sw)
LDI @odci,*+AR3(tms_odci) ;(L/R_sw)
LDI @Vref,*+AR3(tms_Vref) ;(L/R_sw)
LDI @dt,*+AR3(tms_dt) ;(L/R_sw)
LDI @of,*+AR3(tms_of) ;(L/R_sw)
LDI @swf,*+AR3(tms_swf) ;(L/R_sw)
LDI @aci,*+AR3(tms_aci) ;(L/R_sw)
LDI @blk,*+AR3(tms_blk) ;(L/R_sw)
LDI @acs,*+AR3(tms_acs) ;(L/R_sw)
LDI @dcs,*+AR3(tms_dcs) ;(L/R_sw)
LDI @step,*+AR3(tms_step) ;(L/R_sw)
LDI @delay,*+AR3(tms_delay) ;(L/R_sw)
LDI @kc,*+AR3(tms_kc) ;(L/R_sw)
LDI @kcb,*+AR3(tms_kcb) ;(L/R_sw)
LDI @bt,*+AR3(tms_bt) ;(L/R_sw)
LDI @bi,*+AR3(tms_bi) ;(L/R_sw)

```



```

        BR cmd0      ;
        BR cmd0      ;
        RETS

*
* Turning off ARCP
*
cmd0:  LDI 08H,IE      ; Disable interrupts 0,1,2
        LDI @ct_port,AR0 ; Pointer for counter/timer control register
        LDI @clear_main,R0 ;
        STI R0,*AR0      ; Disable all output
        STI R0,*+AR3(tms_outputb);
        LDI 030H,R0      ;
wait20: SUBI 01H,R0      ;
        BNZ wait20      ;
        LDI @reset_out,R0 ;
        STI R0,*AR0      ; Disable all counter/timer output
        CALL init_ct      ;
        LDI 00H,R0      ;
        STI R0,*+AR4(1) ;
        LDI @dp_int,IR0 ;
        STI R0,*+AR4(IR0) ;
        LDI @d_output,AR0 ;
        LDI 0FFFH,R0      ;
        STI R0,*AR0      ;
        LDI @ct_port,AR0 ; Pointer for counter/timer control register
        LDI @reset_out,R0 ;
        STI R0,*AR0      ; Disable all output
        RETS      ;

*
* DC to DC Buck Converter
*
cmd10: LDI 08H,IE      ; Disable interrupts 0,1,2
        LDI @ct_port,AR0 ; Pointer for counter/timer control register
        LDI @clear_main,R0 ;
        STI R0,*AR0      ; Disable all output
        LDI 030H,R0      ;
wait210: SUBI 01H,R0      ;
        BNZ wait210      ;
        LDI @reset_out,R0 ;
        STI R0,*AR0      ; Disable all counter/timer output
        CALL init_ct      ;
        CALL save_setup      ; Save data in 32-bit format
        CALL init_swct      ; Init switching frequency counters
        LDI *+AR3(tms_swp),R0 ;
        FLOAT R0      ;
        LDF @max,R1
        MPYF R0,R1      ;
        STF R1,*+AR3(UMAX) ;
        MPYF @min,R0      ;
        STF R0,*+AR3(UMIN) ;
        LDI *+AR3(tms_Vref),R0 ;
        FLOAT R0      ;
        RND R0      ;
        STF R0,*+AR3(tms_Vref);

```

```

;----- *** Calc limit for Voltage error integrator ***
    LDI  *+AR3(tms_aci),R1 ;
    FLOAT R1 ;
    MPYF @en2,R1 ; scale input to percent
    MPYF 5.0,R1 ;
    RND R1 ;
    STF R1,*+AR3(tms_aci) ;
;----- *** Calc limit for Current error integrator ***
    LDI  *+AR3(tms_dci),R1 ;
    FLOAT R1 ;
    MPYF @en2,R1 ; scale input to percent
    RND R1 ;
    STF R1,*+AR3(tms_dci) ;
*
* start up ramp function
*
    LDI  *+AR3(tms_step),R0;
    STI  R0,*+AR3(stopfreq);
    LDI  000H,R0 ;
    STI  R0,*+AR3(tms_stepx); Startup
    LDI  *+AR3(stopfreq),R0;
    FLOAT R0 ;
    CALL FPINV ;
    MPYF *+AR3(tms_Vref),R0;
    RND R0 ;
    STF R0,*+AR3(vperfreq);
    LDF  0000,R0 ;
    STF R0,*+AR3(tms_Vref);
    LDI  @ctrl,R0 ; Load peripheral bus memory-mapped reg
    LDI  *+AR3(tms_delay),R0;
    MPYI 064H,R0 ;
    STI  R0,*+AR0(28H) ;
    LDI  @tim0ctl,R0 ;
    STI  R0,*+AR0(20H) ; Init internal timer 0
*
* voltage and current scaling
*
    LDI  *+AR3(tms_dcs),R0 ;
    FLOAT R0 ;
    MPYF @inv11bits,R0 ;
    RND R0 ;
    STF R0,*+AR3(tms_dcscale) ;
    LDI  *+AR3(tms_acs),R0 ;
    FLOAT R0 ;
    MPYF @inv11bits,R0 ;
    RND R0 ;
    STF R0,*+AR3(tms_acscale) ;
*
* define hi, hn, hv and T/2
*
    LDI  *+AR3(tms_swf),R0 ; R0 = fsw
    FLOAT R0 ;
    MPYF 2.0,R0 ; R0 = 2*fsw = 2*fsamp
    CALL FPINV ; R0 = Tswp/2

```

; OverCurrent Trip Code

STF R0,*+AR3(tau_2) ; Store T/2

```

    LDI *+AR3(tms_bi),R1 ;
    FLOAT R1 ;
    MPYF @en4,R1 ;
    MPYF R0,R1 ;
    RND R1 ;
    STF R1,*+AR3(K_slave) ;-----> K*T/2
    LDI *+AR3(tms_kc),R1 ;
    FLOAT R1 ;
    MPYF @en4,R1 ; hn
    MPYF R1,R0 ;
    RND R0 ;
    STF R0,*+AR3(hn) ;-----> hn*T/2
    LDI *+AR3(tms_kcb),R0
    FLOAT R0
    MPYF @en4,R0
    RND R0
    STF R0,*+AR3(hv) ;-----> hv
    LDI *+AR3(tms_bt),R0 ;
    FLOAT R0 ;
    MPYF @en4,R0 ;
    RND R0 ;
    STF R0,*+AR3(hi) ;-----> hi
    LDF 0.0,R0 ;
    STF R0,*+AR3(Vdiffa) ; Initialize Vdiff
    STF R0,*+AR3(Vd_inta) ; Initialize Vd_int
    STF R0,*+AR3(Vdiffb) ; Initialize Vdiff
    STF R0,*+AR3(Vd_intb) ; Initialize Vd_int

```

; OverCurrent Trip Code

```

    STF R0,*+AR3(io_m_116) ; Initialize io_m_116
    STF R0,*+AR3(io_s_116) ; Initialize io_s_116
    STF R0,*+AR3(trip_m) ; Initialize trip_m
    STF R0,*+AR3(trip_s) ; Initialize trip_s

```

*****Pulse by pulse Limit to the DAC

```

    LDF *+AR3(tms_acscale),R0 ; acscale is used as current scalefactor
    CALL FPINV ; Generate scalefactor for DC OC Limit
    LDI *+AR3(tms_odci),R1 ; Read in DC OverCurrent Limit
    FLOAT R1
    MPYF R1,R0 ; Scale Threshold Limit Value
    RND R0 ;
    FIX R0 ; R0 = amps_to_counts scalefactor
    XOR mask_dac,R0
    LDI @dac_1,AR0
    STI R0,*AR0 ; Write CurrentLimit to dac_1
    LDI @dac_2,AR0
    STI R0,*AR0 ; write CurrentLimit to dac_2

```



```

*****END
    LDF 0.0, R0
    STF R0, *+AR3(d) ; initialize Master dutycycle
*
*****
    LDI @ct_port, AR0 ; Pointer for counter/timer control register
    LDI 00300H, R0 ; 1100000 (disable phase C)
    STI R0, *AR0 ; Enable all counter/timer output
    STI R0, *+AR3(tms_outputb)
;==*== LDI 010bH, IE ; Enable interrupts 0,1,3,8
    LDI 0030bH, IE ; Enable interrupts 0,1,3,8,9
    LDI 01H, R0
    STI R0, *+AR4(1)
    LDI @dp_int, IR0
    STI R0, *+AR4(IR0)
    RETS
*
save_setup: LDI tblsize, RC ; Init loop counter
            RPTB save_dp ;
            LDI *AR4++(1), R0 ; Start at the top of the dual port memory
            AND 0ffH, R0 ; Mask out all higher bits
            LSH 08H, R0 ; Rotate 8 bits to the left
            LDI *AR4++(1), R1 ; Get LSB
            AND 0ffH, R1
            OR R0, R1
save_dp: STI R1, *AR3++(1) ; Save 32-bit data in internal RAM
        LDI @dp_mem, AR4 ; Reset AR4
        LDI @blk1, AR3 ; Reset AR3
*
        LDI *+AR3(tms_swf), R1;
* BZ init ; Reset if switching frequency is 0
        LDI @swp_const, R0 ; Determine switching period
        CALL divi ;
        STI R0, *+AR3(tms_swp);
**my change to code to have interrupt 1/2 way thru cycle
        LDI 02H, R1 ;old code: LDI 003H, R1
        CALL divi ;
        ADDI 10H, R0 ;
        STI R0, *+AR3(tms_swp_120)
        LDI *+AR3(tms_swp), R0;
        LDI R0, R1 ; Determine ta
        LSH -1H, R0 ;
* LDI *+AR3(tms_btime), R2;
* SUBI R2, R0 ;
        STI R0, *+AR3(tms_ta);
        LDI *+AR3(tms_dt), R0; Determine tb
        LSH 01H, R0 ;
        SUBI R0, R1 ;
        LSH -1H, R1 ;
        FLOAT R1 ;
        RND R1 ;
        STF R1, *+AR3(tms_tb);
        LDI *+AR3(tms_of), R0; Determine stepx
        LDI *+AR3(tms_blk), R1;

```

```

        MPYI R1,R0      ;
*   BZ  init      ;
        LDI  *+AR3(tms_swf),R1;
        CALL divi      ;
        STI  R0,*+AR3(tms_stepx);
        LDI  *+AR3(tms_btime),R1
        STI  R1,*+AR3(tms_tboost);
        LDI  *+AR3(tms_oaci),R2;
        FLOAT R2      ;
        MPYF @en6,R2
        STF  R2,*+AR3(tms_oaci);----STF  R2,*+AR3(tms_ilmin);
        RETS      ;
*
init_swct:LDI  @ct_swfreg,AR0 ; Pointer for switching frequency timer 1
        LDI  *+AR3(tms_swp),R0;
        LDI  *+AR3(tms_swp),R1;
        STI  R0,*+AR0(0) ; Store LSB of counter 0
        STI  R1,*+AR0(1) ; Store LSB of counter 1
        LSH  -08H,R0      ;
        LSH  -08H,R1      ;
        STI  R0,*+AR0(0) ; Store MSB of counter 0
        STI  R1,*+AR0(1) ; Store MSB of counter 1
        NOP
        NOP
        NOP
        LDI  *+AR3(tms_swp_120),R2;
checkout1:LDI  0040H,R0      ;
        STI  R0,*+AR0(3) ; Latch command
        LDI  *+AR0(1),R0      ;
        AND  000FFH,R0      ; Clear all other higher bits
        LDI  *+AR0(1),R1      ;
        LSH  0008H,R1      ;
        AND  00f00H,R1      ;
        OR   R1,R0      ;
        CMPI R2,R0      ;
        BGT  checkout1      ;
        LDI  *+AR3(tms_swp),R0;
        STI  R0,*+AR0(2) ; Store LSB of counter 2
        LSH  -0008H,R0      ;
        STI  R0,*+AR0(2) ; Store MSB of counter 2
*
        LDI  @ct_phasea,AR0 ; Pointer for phase a counter
        LDI  *+AR3(tms_btime),R1 ;
        STI  R1,*+AR0(0) ; Store LSB of counter 0
        LDI  *+AR3(tms_bdly),R1 ;
        ADDI *+AR3(tms_btime),R1 ;
        STI  R1,*+AR0(1) ; Store LSB of counter 1
        LDI  *+AR3(tms_ta),R1 ;
        STI  R1,*+AR0(2) ; Store LSB of counter 2
        LSH  -08H,R1      ;
        STI  R1,*+AR0(2) ; Store MSB of counter 2
*
        LDI  @ct_phaseb,AR0 ; Pointer for phase b counter
        LDI  *+AR3(tms_btime),R1 ;

```

```

    STI R1,*+AR0(0)    ; Store LSB of counter 0
    LDI *+AR3(tms_bdly),R1 ;
    ADDI *+AR3(tms_btime),R1 ;
    STI R1,*+AR0(1)    ; Store LSB of counter 1
    LDI *+AR3(tms_ta),R1 ;
    STI R1,*+AR0(2)    ; Store LSB of counter 2
    LSH -08H,R1        ;
    STI R1,*+AR0(2)    ; Store MSB of counter 2
*
    LDI @ct_phasec,AR0 ; Pointer for phase c counter
    LDI *+AR3(tms_btime),R1 ;
    STI R1,*+AR0(0)    ; Store LSB of counter 0
    LDI *+AR3(tms_bdly),R1 ;
    ADDI *+AR3(tms_btime),R1 ;
    STI R1,*+AR0(1)    ; Store LSB of counter 1
    LDI *+AR3(tms_ta),R1 ;
    STI R1,*+AR0(2)    ; Store LSB of counter 2
    LSH -08H,R1        ;
    STI R1,*+AR0(2)    ; Store MSB of counter 2
    LDI *+AR3(tms_btime),R0 ;
    STI R0,*+AR3(tms_tboost);
    RETS
*
isr_mode: LDI *+AR3(tms_mode),R0 ;
          LDI @mode_ad,R1 ;
          ADDI R1,R0 ;
          BNZ R0 ;
          RETS ;
*
mode_cmd: BR mode0 ; Stop
          BR mode0 ; Test Mode
          BR mode0 ; DC to AC Mode
          BR mode0 ; Motor Control Mode - Forward
          BR mode0 ; Motor Control Mode - Reverse
          BR mode0 ; Actuator Control Mode - Open
          BR mode0 ; Actuator Control Mode - Close
          BR mode0 ; Linear Actuator Mode - Open
          BR mode0 ; Linear Actuator Mode - Close
          BR mode9 ; DC to DC Boost Mode
          BR mode10 ; DC to DC Buck Mode
          BR mode0 ; Stop
          BR mode0 ; Stop
*
mode0: LDI 08H,IE ; Disable interrupts 0,1,2
        LDI @ct_port,AR0 ; Pointer for counter/timer control register
        LDI @clear_main,R0 ;
        STI R0,*AR0 ; Disable all output
        LDI 030H,R0 ;
wait30: SUBI 01H,R0 ;
        BNZ wait30 ;
        LDI @reset_out,R0 ;
        STI R0,*AR0 ; Disable all counter/timer output
        AND 08H,IF ; Clear all pending interrupts 0,1,2
        LDI 00H,R0 ;

```

```

    STI R0,*+AR4(1) ;
    LDI @dp_int,IR0 ;
    STI R0,*+AR4(IR0) ;
    RETS ; Return

*
* test
*
mode9: LDI *+AR3(tms_kc),R7 ;
    RETS ;

*
* DC to DC Buck Converter
*
mode10: LDF *+AR3(tms_Vref),R0 ;RO=Vref
    LDF *+AR3(Vin_inv),R1 ;R1= 1/Vin
    LDF *+AR3(Vout),R2 ;R2= Vout
    LDF *+AR3(Vdiff),R3 ;*R3= Vdiff(n-1)
    LDF *+AR3(iL),R6 ;R6 = iL
    SUBF *+AR3(iout),R6 ;R6 = iL-iout
    MPYF *+AR3(hi),R6 ;R6 = hi(iL-iout)
    MPYF3 R0,R1,R4 ;R4= Dss=Vref/Vin

*s
    STF R4,*+AR3(Dss)

*e
    SUBF3 R0,R2,R5 ;R5= Vdiff(n)=Vout-Vref
    ADDF R5,R3 ;*R3= Vdiff(n)+Vdiff(n-1)
    MPYF *+AR3(hn),R3 ;*R3= Vd_int=KcT/2 [Vdiff(n)+Vdiff(n-1)]
    ADDF *+AR3(Vd_int),R3 ;*R3= Vd_int(n)= Vd_int + Vd_int(n-1)

;----- Limit the Integrator
    LDF R3,R7 ; R7(temp)=Vd_int(n)
    ABSF R7 ;
    CMPF *+AR3(tms_aci),R7 ; CMP[abs(Vd_int(n)) - Iac]
    BLE NoLim10 ; Limit reached stop increasing
    LDF *+AR3(Vd_int),R3 ; R3=Vd_int(old)
NoLim10: NOP ;

;-----
    LDF *+AR3(Dss),R4 ;restore Dss to R4
    SUBF R3,R4 ;R4= D=Dss-Vd_int
    SUBF R6,R4 ;R4 = Dss - Vd_int - hi(iL-iout)
    LDF *+AR3(hv),R6 ;R6 = hv
    MPYF R5,R6 ;R6 = hv(Vout-Vref)
    SUBF R6,R4 ;R4 = Dss - Vd_int - hi(iL-iout) - hv(Vout-Vref)

;----- Limit the duty cycle
HiLim: CMPF @max,R4
    BLE LoLim
    LDF @max,R4
LoLim: CMPF @min,R4
    BGT Same
    LDF @min,R4
Same: NOP

;----- Store Master Dutycycle
    STF R4,*+AR3(d) ; d = Dss - d1

*
    LDI *+AR3(tms_swp),R7
    FLOAT R7

```

```

        MPYF R7,R4
        SUBF R4,R7
;----- Here R7 = (1 - d) to compensate for the PEBB EPLD inversion
        STF R7,*+AR3(count)
*e
        FIX R7
                RETS      ; Return
;=====
;===== EPROM ONLY
        .sect "vecs"      ; Named section
reset    .word  init      ; RS- loads address init to PC
int0     .word  isr0      ; INT0- loads address int0 to PC
int1     .word  isr1      ; INT1- loads address int1 to PC
int2     .word  isr2      ; INT2- loads address int2 to PC
int3     .word  isr3      ; INT3- loads address int3 to PC
*
        .space  4         ; Reserved space
tint0    .word  time0     ; Timer 0 interrupt processing
tint1    .word  time1     ; Timer 1 interrupt processing
        .space  34        ; Reserved space
; end EPROM
;=====
        .data
sram     .word  0080000H   ;==(EPROM)== Beginning of SRAM (init,cmd1)
;==(BOOT)sram .word  0084000H   ;==(BOOT)== Beginning of Sin Table
blk0     .word  0809800H   ; Beginning address of RAM block 0 (init)
blk1     .word  0809C00H   ; Beginning address of RAM block 1 (init,
;
save_setup)
stck     .word  0809F00H   ; Beginning of stack (init)
ctrl     .word  0808000H   ; Pointer for peripheral-bus memory map(init,
;
cmd10,cmd1)
xbus     .word  0000048H   ; Xpansion bus: 2 wait states, external(init)
*                ; RDY not in use (88)
pbus     .word  0000428H   ; Primary bus : 1M bank compare, 1 wait(init)
*                ; states, external RDY not in use
tim0ctl  .word  00003C1H   ; Internal timer 0:1111000001;(301)1100000001
tim1ctl  .word  00003C1H   ; Internal timer 1: 1111000001;(301) 1100000001
tim1prd  .word  007A120H   ;7A120H=500000d,10MHz, 50%duty =50ms*2=100ms
wait4t   .word  0000100H   ; (cmd10,cmd1)
*
*
*
dp_mem   .word  0100000H   ; Pointer for dual port memory (command reg)
; (init,
save_setup)
dp_int   .word  00003FEH   ; Pointer for setting interrupt flag (cmd10,
;
isr_mode,cmd1)
dp_cint  .word  00003FFH   ; Pointer for clearing interrupt flag (init)
*
*

```

```

tms_oaci .set 0000001H ; Ac trip current level (save_setup,
;

set_oc,cmd29)
tms_acv .set 0000002H ; test
tms_bdly .set 0000003H ; Boost delay (init_swct)
tms_btime .set 0000004H ; Boost time (save_setup, init_swct)
tms_dci .set 0000005H ; Dc current
tms_odci .set 0000006H ; Dc trip current level(set_oc)
tms_Vref .set 0000007H ; Dc voltage
tms_dt .set 0000008H ; Deadtime (save_setup)
tms_of .set 0000009H ; Ac frequency (save_setup)
tms_swf .set 000000aH ; Switching frequency (save_setup)
tms_aci .set 000000bH ; Ac current
tms_blk .set 000000cH ; Block size (cmd10,save_setup,
;

sine_tbl,cmd1)
tms_acs .set 000000dH ; current sensor
tms_dcs .set 000000eH ; voltage sensor (set_oc,cmd10,cmd1)
tms_step .set 000000fH ; Step
tms_delay .set 0000010H ; Delay
tms_swp .set 0000011H ; Switching period (init,cmd10,save_setup,
;

init_swct,cmd1)
tms_stepx .set 0000012H ; Step(cmd10,save_setup,isr0,isr1,isr2,cmd26)
tms_ta .set 0000013H ; ta const(init_swct,model0,save_setup,model1)
tms_tb .set 0000014H ; tb const(init,cmd10,save_setup,model0,cmd1,
;

model)
tms_kc .set 0000015H ; (init_pid,model0)
tms_kcb .set 0000016H
tms_bt .set 0000017H ; (init_pid,model0)
tms_bi .set 0000018H ; (init_pid,model0)
tms_mode .set 0000019H ; Mode (isr_mode,cmd27)
tms_swp_120 .set 000001aH ; (init_swct,save_setup)
tms_command .set 000001bH ;command (L/R_sw)
L_R_posit .set 000001cH ;Local/remote sw posit (L/R_sw)
Vref_desired .set 000001dH ;front panel desired Vref (L/R_sw,time1)
UMIN .set 000001eH ; (init_pid,model0)
UMAX .set 000001fH ; (init_pid,model0)
Vdiff .set 0000020H ; Vout-Vref
Vd_int .set 0000021H ; integral of Vdiff
hn .set 0000022H ;
hv .set 0000023H
hi .set 0000024H
iL .set 0000025H
iout .set 0000026H
Vdiffa .set 0000027H
Vdiffb .set 0000028H
Vd_inta .set 0000029H
Vd_intb .set 000002aH
Vout .set 000002fH ; DC input (model0)
Vin_inv .set 0000030H ; "
DUTY .set 0000031H
vperfreq .set 0000033H ; Volt per frequency ratio

```

```

stopfreq .set 0000034H ; Target frequency
stopvolt .set 0000035H ; Target voltage
tms_invdv .set 0000036H ; (init,cmd10,cmd1)
*s
Dss .set 0000037h
d .set 0000038h
count .set 0000039h
*e
tms_tboost .set 000003aH ; (save_setup,init_swct,isr0,isr1,isr2,cmd28)
tms_acscale .set 000003bH
tms_dcscscale .set 000003cH ; (cmd10,init,cmd1)
tms_outputb .set 000003eH ; (cmd10,cmd1)
tms_ilmin .set 000003fH ; (save_setup,cmd29)
tblsize .set 000001aH ; Setup table size (save_setup)
K_slave .set 0000040H
iL_slave .set 0000041H ;
iL_master .set 0000042H ;
io_slave .set 0000043H ;
io_master .set 0000044H ;
Vin .set 0000045H ;

```

;OverCurrent Trip Code

```

trip_m .set 0000046H ;
trip_s .set 0000047H ;
io_m_116 .set 0000048H ;
io_s_116 .set 0000049H ;
tau_2 .set 000004aH ;

```

*

*

```

ct_swfreq .word 0804000H ; Switching freq timer (init_ct,init_swct)
ct_port .word 0804100H ; Timer control register (ct_port,init_ct,

```

;

cmd10,isr_mode,cmd1)

```

ct_phasea .word 0804200H ; Phase A timer
ct_phaseb .word 0804300H ; Phase B timer (init_ct,init_swct,isr1)
ct_phasesc .word 0804400H ; Phase C timer (init_ct,init_swct,isr2)
d_output .word 0804500H ; General purpuse D/O port (init,cmd1)
d_input .word 0804600H ; General purpuse digital input port
inputcs .word 0804900H ; Input voltage and current ADC (init)
acs .word 0804a00H ; Phase a output V & I ADC (isr1,isr2)
bcs .word 0804b00H ; Phase b output V & I ADC (isr0,isr2)
ccs .word 0804c00H ; Phase c output V & I ADC (isr0,isr1)
dac_1 .word 0804700H ; Digital to Analog converter 1
dac_2 .word 0804800H ; Digital to Analog converter 2

```

*

```

cmd_ad .int startcmd ; (read_cmd)
mode_ad .int mode_cmd ; (isr_mode)

```

*

```

mask_int0 .set 0000001H ; Set external interrupt 0 (isr0)
mask_int1 .set 0000002H ; Set external interrupt 1 (isr1)
mask_int2 .set 0000004H ; Set external interrupt 2 (isr2)

```

```

mask_int3 .set 0000008H ; Set external interrupt 3 (isr3)
mask_timer0 .set 0000100H ; Set internal timer 0 interrupt
mask_timer1 .set 0000200H ; Set internal timer 1 interrupt
mask_dac .set 0000800H ; allow 2's comp numbers in dac

```

*

*

```

clear_main .word 0004444H ; (cmd10,isr_mode,cmd1)
reset_out .word 000ffffH ; (init,cmd10,isr_mode,cmd1)

```

*

* Define default values L/R_sw

*

```

oaci .word 300
acv .word 120
bdly .word 10
btime .word 4
dci .word 10
odci .word 200
Vref .word 43
dt .word 14
of .word 60
swf .word 20000
aci .word 10
blk .word 2000
acs .word 50
dcs .word 500
step .word 50
delay .word 10000
kc .word 17333
kcb .word 9
bt .word 105
bi .word 2
L_R_posit .word 00H
command .word 10
mode .word 10

```

* Define constants

*

```

swp_const .word 10000000 ; (save_setup)
inv11bits .float 0.00048828125 ; (cmd10,isr0,cmd1)
mil .float 0.001 ; (init_pid)
en7 .float 0.0000001 ;
en6 .float 0.000001
en5 .float 0.00001
en4 .float 0.0001
en3 .float 0.001
en2 .float 0.01
en1 .float 0.1
AVE .float 0.2
max .float 0.95
min .float 0.05

```

; OverCurrent Trip Code


```

full      .float 116.0
limit     .float 58.0
*****
*
;==(BOOT)cmd      .usect "dualport",10000h ;==(BOOT)==
cmd      .usect "dualport",10000h ;==(EPROM)==
*
ctio      .usect "xbus",2000h
lookup    .usect "ram1",400h
variable  .usect "ram2",400h
*
;=====
;=====
*
*
* isr0: SSCM SLAVE UNIT interrupt service routine
*
;==(BOOT)          .sect    "isr0"      ;==(BOOT)== Named Section
isr0:  NOP          ;==(EPROM)==
      PUSH ST      ; Save registers
      PUSH IR1      ;
      PUSH R7       ;
      PUSHF R7      ;
      PUSH R6       ;
      PUSHF R6      ;
      PUSH R5       ;
      PUSHF R5      ;
      PUSH R4       ;
      PUSHF R4      ;
      PUSH R3       ;
      PUSHF R3      ;
      PUSH R2       ;
      PUSHF R2      ;
      PUSH R1       ;
      PUSHF R1      ;
      PUSH R0       ;
      PUSHF R0      ;
      PUSH AR0      ;
      PUSH AR1      ;
      PUSH AR2      ;
      PUSH AR3      ;
      PUSH AR4      ;
      PUSH AR5      ;
      PUSH AR6      ;
      PUSH AR7      ;
*
      LDI           0A0H,R7
wait00: SUBI      01H,R7
      BNZ          wait00
*
      LDI           @inputcs,AR0      ; Pointer for DC ADC
      LDI           @acs,AR1
LDI      @bcs,AR2      ;
      LDI           @ccs,AR3

```

```

*
        LDI            *AR0,R0                ; start conversion
        NOP
        NOP
        LDI            *AR1,R1
        NOP
        NOP
        LDI            *AR2,R2
        NOP
        NOP
        LDI            *AR3,R3
        NOP
        NOP
        LDI            00FH,R7
wait0:  SUBI            01H,R7
        BNZ            wait0
*
* STORE SAMPLED VOLTAGES AND CURRENTS
*
        LDI            *AR0,R0                ; READ.. Vinput(LSB)
AND iL_slave(MSB)
        NOP
        NOP
        LDI            *AR1,R1                ; READ.. Voutput(LSB) AND io_slave(MSB)
        NOP
        NOP
        LDI            *AR2,R2                ; READ.. iL_slave(LSB) AND iL_master(MSB)
        NOP
        NOP
        LDI            *AR3,R4                ; READ.. io_slave(LSB) AND io_master(MSB)
        NOP
        NOP
        A2Dfltr R0, R7        ; R0= Vinput(LSB), R7=iL_slave(MSB)
        A2Dfltr R1, R7        ; R1= Voutput(LSB), R7=io_slave(MSB)
        A2Dfltr R2, R3        ; R2=iL_slave(LSB), R3=iL_master(MSB)
        A2Dfltr R4, R5        ; R4=io_slave(LSB), R5=io_master(MSB)
*****
        LDI            *AR0,AR4                ; READ.. Vinput(LSB) AND
iL_slave(MSB)
        NOP
        NOP
        LDI            *AR1,AR5                ; READ.. Voutput(LSB) AND io_slave(MSB)
        NOP
        NOP
        LDI            *AR2,AR6                ; READ.. iL_slave(LSB) AND iL_master(MSB)
        NOP
        NOP
        LDI            *AR3,AR7                ; READ.. io_slave(LSB) AND io_master(MSB)
        NOP
        NOP
*****Process and Accumulate Data
        LDI            AR4, R6
        A2Dfltr R6, R7        ; R6= Vinput(LSB), R7=iL_slave(MSB)
        ADDF            R6, R0                ; R0= Vinput (2)

```

```

;----
LDI  AR5, R6
      A2Dfltr R6, R7      ; R6= Voutput(LSB), R7=io_slave(MSB)
      ADDF    R6, R1      ; R1= Voutput (2)
;----
LDI  AR6, R6
      A2Dfltr R6, R7      ; R6=iL_slave(LSB), R7=iL_master(MSB)
      ADDF    R6, R2      ; R2 = iL_slave (2)
      ADDF    R7, R3      ; R3 = iL_master (1)
;----
LDI  AR7, R6
      A2Dfltr R6, R7      ; R6=io_slave(LSB), R7=io_master(MSB)
      ADDF    R6, R4      ; R4 = iout_slave (2)
      ADDF    R7, R5      ; R5 = iout_master (1)
*****
      LDI      *AR0,AR4    ; READ.. Vinput(LSB) AND iL_slave(MSB)
      NOP
      NOP
      LDI      *AR1,AR5    ; READ.. Voutput(LSB) AND io_slave(MSB)
      NOP
NOP
      LDI      *AR2,AR6    ; READ.. iL_slave(LSB) AND iL_master(MSB)
      NOP
      NOP
      LDI      *AR3,AR7    ; READ.. io_slave(LSB) AND io_master(MSB)
      NOP
      NOP
*****Process and Accumulate Data
LDI  AR4, R6
      A2Dfltr R6, R7      ; R6= Vinput(LSB), R7=iL_slave(MSB)
      ADDF    R6, R0      ; R0= Vinput (2)
;----
LDI  AR5, R6
      A2Dfltr R6, R7      ; R6= Voutput(LSB), R7=io_slave(MSB)
      ADDF    R6, R1      ; R1= Voutput (2)
;----
LDI  AR6, R6
      A2Dfltr R6, R7      ; R6=iL_slave(LSB), R7=iL_master(MSB)
      ADDF    R6, R2      ; R2 = iL_slave (2)
      ADDF    R7, R3      ; R3 = iL_master (1)
;----
LDI  AR7, R6
      A2Dfltr R6, R7      ; R6=io_slave(LSB), R7=io_master(MSB)
      ADDF    R6, R4      ; R4 = iout_slave (2)
      ADDF    R7, R5      ; R5 = iout_master (1)
*****
      LDI      *AR0,AR4    ; READ.. Vinput(LSB) AND
iL_slave(MSB)
      NOP
      NOP
      LDI      *AR1,AR5    ; READ.. Voutput(LSB) AND io_slave(MSB)
      NOP
NOP
      LDI      *AR2,AR6    ; READ.. iL_slave(LSB) AND iL_master(MSB)

```

```

                NOP
                NOP
LDI      *AR3,AR7      ; READ.. io_slave(LSB) AND io_master(MSB)
                NOP
                NOP
*****Process and Accumulate Data
LDI      AR4, R6
                A2Dfltr R6, R7      ; R6= Vinput(LSB), R7=iL_slave(MSB)
                ADDDF   R6, R0      ; R0= Vinput (2)
;-----
LDI      AR5, R6
                A2Dfltr R6, R7      ; R6= Voutput(LSB), R7=io_slave(MSB)
                ADDDF   R6, R1      ; R1= Voutput (2)
;-----
LDI      AR6, R6
                A2Dfltr R6, R7      ; R6=iL_slave(LSB), R7=iL_master(MSB)
                ADDDF   R6, R2      ; R2 = iL_slave (2)
                ADDDF   R7, R3      ; R3 = iL_master (1)
;-----
LDI      AR7, R6
                A2Dfltr R6, R7      ; R6=io_slave(LSB), R7=io_master(MSB)
                ADDDF   R6, R4      ; R4 = iout_slave (2)
                ADDDF   R7, R5      ; R5 = iout_master (1)
*****
iL_slave(MSB)
                LDI      *AR0,AR4      ; READ.. Vinput(LSB) AND
                NOP
                NOP
                LDI      *AR1,AR5      ; READ.. Voutput(LSB) AND io_slave(MSB)
                NOP
NOP
                LDI      *AR2,AR6      ; READ.. iL_slave(LSB) AND iL_master(MSB)
                NOP
                NOP
LDI      *AR3,AR7      ; READ.. io_slave(LSB) AND io_master(MSB)
                NOP
                NOP
*****Process and Accumulate Data
LDI      AR4, R6
                A2Dfltr R6, R7      ; R6= Vinput(LSB), R7=iL_slave(MSB)
                ADDDF   R6, R0      ; R0= Vinput (2)
;-----
LDI      AR5, R6
                A2Dfltr R6, R7      ; R6= Voutput(LSB), R7=io_slave(MSB)
                ADDDF   R6, R1      ; R1= Voutput (2)
;-----
LDI      AR6, R6
                A2Dfltr R6, R7      ; R6=iL_slave(LSB), R7=iL_master(MSB)
                ADDDF   R6, R2      ; R2 = iL_slave (2)
                ADDDF   R7, R3      ; R3 = iL_master (1)
;-----
LDI      AR7, R6
                A2Dfltr R6, R7      ; R6=io_slave(LSB), R7=io_master(MSB)
                ADDDF   R6, R4      ; R4 = iout_slave (2)

```

```

        ADDF      R7, R5                                ; R5 = iout_master (1)
*****
        POP  AR7      ;
        POP  AR6      ;
        POP  AR5      ;
        POP  AR4      ;
        POP  AR3      ;
        POP  AR2      ;
        POP  AR1      ;
;----- Calculate System Voltages
        MPYF  @AVE,R0
        MPYF  *+AR3(tms_dcscale),R0 ;
        RND      R0
        STF      R0,*+AR3(Vin)                ; STORE Input voltage Vin
;-----
        MPYF  @AVE,R1
        MPYF  *+AR3(tms_dcscale),R1 ;
        RND      R1
        STF      R1,*+AR3(Vout)                ; STORE Output voltage Vout
;-----
        MPYF  @AVE,R2
        MPYF  *+AR3(tms_acscale),R2
        RND  R2
        STF  R2,*+AR3(iL_slave)
;-----
        MPYF  @AVE,R3
        MPYF  *+AR3(tms_acscale),R3
        RND  R3
        STF  R3,*+AR3(iL_master)
;-----
        MPYF  @AVE,R4
        MPYF  *+AR3(tms_acscale),R4
        RND  R4
        STF  R4,*+AR3(io_slave)
;-----
        MPYF  @AVE,R5
        MPYF  *+AR3(tms_acscale),R5
        RND  R5
        STF  R5,*+AR3(io_master)

*****
; OverCurrent Trip Code

        LDF  @full,R7      ; R7=116.0
        SUBF R7,R5         ; R5=io_master(n)-116.0
        LDF  *+AR3(io_m_116),R7 ; R7=io_master(n-1)-116.0
        STF  R5,*+AR3(io_m_116) ; Save io_master(n) for next pass
        ADDF R5,R7         ; R7=Sum of n and n-1
        LDF  *+AR3(tau_2),R5 ; R5=T/2
        MPYF R5,R7         ; R7=T/2(n + n-1)
        LDF  *+AR3(trip_m),R5 ; R5 previous integral total
        ADDF R5,R7         ; R7 Total integral value
        BN   clrtipm       ; Assuring non-negative integral
        STF  R7,*+AR3(trip_m) ; Stores trip_m(n) for next pass

```

```

LDI @cmd_ad,R5 ; Jump target if needed for shutdown
LDF @limit,R6 ; R6=58.0 integral limit
SUBF R6,R7 ;
BNN R5 ; Shuts down Bucks
BR iokm ; Branch to output current okay
clrtripm: LDF 0.0,R5 ;
STF R5,*+AR3(trip_m) ; Resets integral if negative
iokm: LDF *+AR3(io_master),R5 ; Resets R5 to io_master

```

```

LDF @full,R7 ; R7=116.0
SUBF R7,R4 ; R4=io_slave(n)-116.0
LDF *+AR3(io_s_116),R7 ; R7=io_slave(n-1)-116.0
STF R4,*+AR3(io_s_116) ; Save io_slave(n) for next pass
ADDF R4,R7 ; R7=Sum of n and n-1
LDF *+AR3(tau_2),R4 ; R4=T/2
MPYF R4,R7 ; R7=T/2(n + n-1)
LDF *+AR3(trip_s),R4 ; R4 previous integral total
ADDF R4,R7 ; R7 Total integral value
BN clrtrips ; Assuring non-negative integral
STF R7,*+AR3(trip_s) ; Stores trip_s(n) for next pass
LDI @cmd_ad,R4 ; Jump target if needed for shutdown
LDF @limit,R6 ; R6=58.0 integral limit
SUBF R6,R7 ;
BNN R4 ; Shuts down Bucks
BR ioks ; Branch to output current okay
clrtrips: LDF 0.0,R4 ;
STF R4,*+AR3(trip_s) ; Resets integral if negative
ioks: LDF *+AR3(io_slave),R4 ; Resets R4 to io_slave

```

```

;-----
ADDF3 R4, R5, R0
RND R0
STF R0,*+AR3(iout) ; STORE THE TOTAL OUTPUT CURRENT iout
;-----
ADDF3 R2, R3, R0
RND R0
STF R0,*+AR3(iL) ; STORE THE TOTAL Inductor CURRENT iL
;----- Calculate ---> Iout_error
SUBF R4, R5 ; R5= Vdiff(n) == (io_master - io_slave)
;-----
LDF *+AR3(Vin),R0
CALL FPINV ;
RND R0 ;
STF R0,*+AR3(Vin_inv) ; STORE 1/Vin
;-----
LDF *+AR3(Vdiffa), R3 ; Prepare for Trapzd integration....
STF R3, *+AR3(Vdiff) ; by loading old (n-1) values into Vdiff
LDF *+AR3(Vd_inta), R0 ; and Vd_int
STF R0, *+AR3(Vd_int) ;
;----- TRAPZD INTEGRATION from Mode 10 Routine
LDF *+AR3(Vdiff),R3 ; *R3= Vdiff(n-1)
ADDF R5,R3 ; R3= Vdiff(n)+Vdiff(n-1)

```

```

                MPYF  *+AR3(K_slave),R3                ; R3= Vd_int=K*T/2 [Vdiff(n)+Vdiff(n-1)]
                ADDF  *+AR3(Vd_int),R3                  ; R3= Vd_int(n) = Vd_int + Vd_int(n-1)
;----- Limit the Integrator
    LDF  R3, R7                ; R7(temp)=Vd_int(n)
    ABSF R7                    ;
    CMPF *+AR3(tms_dci),R7     ; [abs(Vd_int(n)) - Idc]
    BLE  NoLim0                ; Limit reached stop increasing
    LDF  *+AR3(Vd_int),R3      ; R3=Vd_int(old)
NoLim0: NOP                    ;
;----- Save Integ quantities for next time
    RND      R5
    STF      R5,*+AR3(Vdiffa)
    RND      R3
    STF      R3,*+AR3(Vd_inta)
;----- Get Master DutyCycle *****
*
    LDF      *+AR3(Vdiffb),R0
    STF      R0,*+AR3(Vdiff)
    LDF      *+AR3(Vd_intb),R0
    STF      R0,*+AR3(Vd_int)
    CALL  isr_mode                ;
;----- Save Integ quantities for next time
    RND      R5
    STF      R5,*+AR3(Vdiffb)
    RND      R3
    STF      R3,*+AR3(Vd_intb)
;----- Write the Master Duty to A phase CTC (S1) *****
    LDI  @ct_phasea,AR0 ; Pointer for phase A counter
    STI  R7,*+AR0(2)    ; Store LSB of counter 2
    LSH  -08H,R7        ;
    STI  R7,*+AR0(2)    ; Store MSB of counter 2
*
;----- UPDATE SLAVE DUTY CYCLE *****
    LDF  *+AR3(d), R4      ; R4 = (Dss - d1) --> master dutycycle
    ADDF *+AR3(Vd_inta), R4 ; R4 = (Dss - d1) + Vd_int
*
    LDF *+AR3(tms_oaci),R0 ;R0=kp (GUI value oaci)
    MPYF *+AR3(Vdiffa), R0 ;R0=kp(iomaster-ioslave)
    ADDF R0, R4            ;R4=(Dss-d1)+Vdint+kp(iomaster-ioslave)
*
    ; = Master_Duty+integ(ki*io_err)+kp(io_err)
;----- Limit the duty cycle
HiLim0: CMPF @max,R4
        BLE  LoLim0
        LDF  @max,R4
LoLim0: CMPF @min,R4
        BGT  Same0
        LDF  @min,R4
Same0: NOP
;-----
    LDI      *+AR3(tms_swp),R7                ;
    FLOAT R7                                ;
    MPYF R7,R4                                ;
    SUBF R4,R7                                ; R7= duty = tms_swp - (R4* tms_swp)
;----- Here R7 = (1 - d) to compensate for the PEBB EPLD inversion

```

```

    FIX          R7          ;
;----- Write the Slave Duty to B phase CTC (S1) *****
    LDI @ct_phaseb,AR0 ; Pointer for phase B counter
    STI R7,*+AR0(2) ; Store LSB of counter 2
    LSH -08H,R7 ;
    STI R7,*+AR0(2) ; Store MSB of counter 2
;*****
; Overtem/undervoltage protection (L/R_sw)
; Gen I/O word bit0 = overtemp slave, bit1 = overtemp master
; bit2 = control voltage slave, bit3 = control voltage master
    LDI @d_output,AR0 ;set pointer to Gen I/O
    STI *AR0,R0 ;R0=gen_I/O word
    AND 000fH,R0 ;mask all but 4 lsbs
    LDI @cmd_ad,R4 ; Jump target if needed for shutdown
    CMPI 0fH,R0 ;see if word is good
    BNE R4 ; Shuts down Bucks
;*****
    POP AR0 ;
    POPF R0 ;
    POP R0 ;
    POPF R1 ;
    POP R1 ;
    POPF R2 ;
    POP R2 ;
    POPF R3 ;
    POP R3 ;
    POPF R4 ;
    POP R4 ;
    POPF R5 ;
    POP R5 ;
    POPF R6 ;
    POP R6 ;
    POPF R7 ;
    POP R7 ;
    POP IR1 ;
    POP ST ;
    ANDN mask_int0,IF ; Clear interrupt 0
    RETI ; Return and enable interrupt
;=====
;=====
*
* isr1: SSCM MASTER UNIT --- interrupt service routine
*
;==(BOOT) .sect "isr1" ;==(BOOT)== Named Section
isr1: NOP ;==(EPROM)==
    ANDN mask_int1,IF ; Clear interrupt 1
    RETI ; Return and enable interrupt
;=====
;=====
*
* isr2: Phase C interrupt service routine
*
;==(BOOT) .sect "isr2" ;==(BOOT)== Named Section
isr2: NOP ;==(EPROM)==

```



```

        ANDN mask_int2,IF ; Clear interrupt 2
        RETI                ; Not Used
=====
*
*
*
* irs3: Dual port memory interrupt service routine
*
;==(BOOT)                .sect    "irs3"        ;==(BOOT)== Named Section
irs3:  NOP                ;==(EPROM)==
        PUSH ST            ; Save registers
        PUSH DP            ;
        PUSH IR1           ;
        PUSH R7            ;
        PUSHF R7           ;
        PUSH R6            ;
        PUSHF R6           ;
        PUSH R5            ;
        PUSHF R5           ;
        PUSH R4            ;
        PUSHF R4           ;
        PUSH R3            ;
        PUSHF R3           ;
        PUSH R2            ;
        PUSHF R2           ;
        PUSH R1            ;
        PUSHF R1           ;
        PUSH R0            ;
        PUSHF R0           ;
*
        LDI @dp_cint,IR0 ;
        LDI *+AR4(IR0),R0 ; Clear interrupt
        CALL read_cmd      ;
        ANDN mask_int3,IF ; Clear interrupt 3
*
        POPF R0            ;
        POP R0             ;
        POPF R1            ;
        POP R1             ;
        POPF R2            ;
        POP R2             ;
        POPF R3            ;
        POP R3             ;
        POPF R4            ;
        POP R4             ;
        POPF R5            ;
        POP R5             ;
        POPF R6            ;
        POP R6             ;
        POPF R7            ;
        POP R7             ;
        POP IR1            ;
        POP DP             ;
        POP ST             ;
        NOP

```

```

NOP
    RETI                ; Return and enable interrupt
*=====
*
* timer0: Startup timer
*
;(BOOT)                .sect    "time0"        ;==(BOOT)== Named Section
time0: NOP              ;==(EPROM)==
    PUSH R0              ;
    PUSHF R0             ;
    PUSH AR0             ;
    LDI  *+AR3(tms_stepx),R0;
    ADDI 01H,R0          ;
    STI R0,*+AR3(tms_stepx);
    CMPI *+AR3(stopfreq),R0;
    BLE looptimer0      ;
    LDI 000H,R0          ;
    LDI @ctrl,AR0        ;
    STI R0,*+AR0(20H)    ; Clear counter
    ANDN mask_timer0,IE ; Disable timer interrupt
    POP AR0              ;
    POPF R0              ;
    POP R0               ;
    RETI                 ;
looptimer0: LDF *+AR3(vperfreq),R0;
    ADDF *+AR3(tms_Vref),R0;
    RND R0               ;
    STF R0,*+AR3(tms_Vref);
    POP AR0              ;
    POPF R0              ;
    POP R0               ;
    RETI                 ;
*=====
*
* timer1: 100ms Timer
*
;==(BOOT) .sect    "time1"        ;==(BOOT)== Named Section
time1: NOP              ;==(EPROM)==
    PUSH ST
    PUSH DP
    PUSH IR1
    PUSH R7
    PUSHF R7
    PUSH R6              ;
    PUSHF R6
    PUSH R6              ;
    PUSHF R5
    PUSH R4              ;
    PUSHF R4
    PUSH R3              ;
    PUSHF R3
    PUSH R2              ;

```

```

    PUSHF R2
    PUSH    R1    ;
    PUSHF R1
    PUSH    R0    ;
    PUSHF R0
    PUSH    AR0   ;
*

    LDI *+AR3(L_R_posit),R1 ;get previous L/R posit
    LDI @d_output,AR0      ;set pointer to Gen I/O
    STI *AR0,R0            ;R0=gen_I/O word
    AND 0010H,R0           ;mask all but L/R sw posit (bit4)
    CMPI 00H,R0            ;if 0, in remote
    BEQ remote

local: LDI 303H,IE         ;disables int3 in local mode
       CMP R0,R1          ;see if sw same as last interrupt
       BEQ update_vref    ;if in local, update vref and end_int

       ;if sw is now in L from R, shutdown and restart in local mode
       STI R0,*+AR3(L_R_posit) ;save current sw posit
       call cmd0            ;to shutdown unit
restart: call read_cmd      ;to restart
       BR End_int

update_vref: LDF R0,*+AR3(tms_Vref) ;get Vref from memory
            LDI @adc1_cs, AR0      ;pointer to ADC for front panel
            LDI *AR0,R1            ;loads old data,init's new read
            NOP
            NOP                   ;2 nops for delay
            LDI *AR0,R1            ;read in new Vref from front panel
            A2Dfltr R1,R7          ;to extract front panel voltage if needed
            MPYF *+AR3(tms_dcscaler),R1 ;scale the word to make actual voltage
            STF R1,*+AR3(Vref_desired) ;store front panel as desired Vref
            SUBF3 R0,R1,R2         ;R2=R1-R0 (V_desired-Vref)
            CMPF 10.0,R2          ;see if greater than 10V increase
            BLE no_step

step: LDI 00H,*+AR3(tms_stepx) ;set counter to zero
      MPYF @en1,R2             ;R2=voltage difference/10
      FIX R2                   ;make R2 an integer
      LDI R2,*+AR3(stopfreq) ;use R2 as number of steps required
      LDF 10.0,*+AR3(vperfreq) ;10v is the step size
      LDI @ctrl,AR0           ;
      LDI @tim1prd,R0          ;load 100ms period
      STI R0,*+AR0(28H)        ;load in timer0
      LDI @tim0ctrl,R0         ;
      STI R0,*+AR0(30H)        ;init timer0 for step
      BR End_int

no_step: LDF R1,*+AR3(tms_Vref) ;if no step req'd, save front panel as Vref
        BR End_int

remote: LDI 30bH,IE           ;enable intr 0,1,3,8,9
        CMP R0,R1            ;see if previous sw posit matches present

```

BEQ End_int ;if still in remote, end intr

;if sw changed from local and now in remote, shutdown and wait for PC command

STI R0,*+AR3(L_R_posit) ;store present sw posit for later

call cmd0 ;to shutdown bucks

End_int: ANDN mask_timer1,IF; RESET timer interrupt Flag

POP AR0

POPF R0

POP R0

POPF R1

POP R1

POPF R2

POP R2

POPF R3

POP R3

POPF R4

POP R4

POPF R5

POP R5

POPF R6

POP R6

POPF R7

POP R7

POP IR1

POP DP

POP ST

RETI

*

.end

APPENDIX C. ARCP CLOSED-LOOP CONTROL CODE

```

*****
*
*       NPS POWER LAB
*       TMS320C30 CONTROL CODE
*       BY TUAN DUONG NSWC
*
*       MODIFIED FOR CLOSED-LOOP CONTROL OF THE ARCP
*       BY DAVID FLOODEEN
*****
*
*       .title  "PEBB"
*
*       .global reset,init
*       .global int0,int1,int2,int3
*       .global tint0
*       .global isr0,isr1,isr2,isr3
*       .global time0
*       .global SIN,FPINV,divi
*
*       .sect  "vecs"          ; Named section
reset .word  init             ; RS- loads address init to PC
int0  .word  isr0             ; INT0- loads address int0 to PC
int1  .word  isr1             ; INT1- loads address int1 to PC
int2  .word  isr2             ; INT2- loads address int2 to PC
int3  .word  isr3             ; INT3- loads address int3 to PC
*
*       .space  4              ; Reserved space
tint0 .word  time0            ; Timer 0 interrupt processing
tint1 .word  time1            ; Timer 1 interrupt processing
*       .space  33             ; Reserved space
*
*       .data
sram  .word  0080000H          ; Beginning address of SRAM
blk0  .word  0809800H          ; Beginning address of RAM block 0
blk1  .word  0809C00H          ; Beginning address of RAM block 1
stck  .word  0809F00H          ; Beginning of stack
ctrl  .word  0808000H          ; Pointer for peripheral-bus memory map
xbus  .word  0000048H          ; Xpansion bus: 2 wait states, external RDY
                                   ; not in use (88)
pbus  .word  0000428H          ; Primary bus : 1M bank compare, 1 wait states,
                                   ; external RDY not in use
tim0ctl .word  00003C1H        ; Internal timer 0: 1111000001; (301) 1100000001
tim0prd .word  0000064H        ; 40H1 (10us)
tim1ctl .word  00003C1H        ; Internal timer 1: 1111000001; (301) 1100000001
tim1prd .word  0004E20H        ; 40H1 (2ms)
wait4t .word  0000100H        ;
*
dp_mem .word  0100000H          ; Pointer for dual port memory (command reg)
dp_int .word  00003FEH          ; Pointer for setting interrupt flag
dp_cint .word  00003FFH          ; Pointer for clearing interrupt flag

```

```

dp_cmd .set 0000000H ; Command register
dp_oaci .set 0000002H ; Ac trip current level
dp_acv .set 0000004H ; Ac voltage
dp_bdly .set 0000006H ; Boost delay
dp_btime .set 0000008H ; Boost time
dp_dci .set 000000aH ; Dc current
dp_odci .set 000000ch ; Dc trip current level
dp_dcv .set 000000eH ; Dc voltage
dp_dt .set 0000010H ; Deadtime
dp_of .set 0000012H ; Ac frequency
dp_swf .set 0000014H ; Switching frequency
dp_aci .set 0000016H ; Ac current
dp_blk .set 0000018H ; Block size
dp_acs .set 000001aH ; Ac sensor
dp_dcs .set 000001cH ; Dc sensor
dp_step .set 000001eH ; Step
dp_delay .set 0000020H ; Delay
dp_swp .set 0000022H ; Switching period
dp_stepx .set 0000024H ; Step
dp_ta .set 0000026H ; ta constant
dp_tb .set 0000028H ; tb constant
dp_kc .set 000002aH ;
dp_kcb .set 000002cH ;
dp_bt .set 000002eH ;
dp_bi .set 0000030H ;
dp_mode .set 0000032H ; Mode
*
*
tms_cmd .set 0000000H ; Command register
tms_oaci .set 0000001H ; Ac trip current level
tms_acv .set 0000002H ; Ac voltage
tms_bdly .set 0000003H ; Boost delay
tms_btime .set 0000004H ; Boost time
tms_dci .set 0000005H ; Dc current
tms_odci .set 0000006h ; Dc trip current level
tms_dcv .set 0000007H ; Dc voltage
tms_dt .set 0000008H ; Deadtime
tms_of .set 0000009H ; Ac frequency
tms_swf .set 000000aH ; Switching frequency
tms_aci .set 000000bH ; Ac current
tms_blk .set 000000cH ; Block size
tms_acs .set 000000dH ; Ac sensor
tms_dcs .set 000000eH ; Dc sensor
tms_step .set 000000fH ; Step
tms_delay .set 0000010H ; Delay
tms_swp .set 0000011H ; Switching period
tms_stepx .set 0000012H ; Step
tms_ta .set 0000013H ; ta constant
tms_tb .set 0000014H ; tb constant
tms_kc .set 0000015H ;
tms_kcb .set 0000016H ;
tms_bt .set 0000017H ;
tms_bi .set 0000018H ;
tms_mode .set 0000019H ; Mode

```

```

tms_swp_120 .set 000001aH ;
tms_cos .set 000001bH ;offset for pointer to cos in sin table
*tms_23 .set 000001cH ;
tms_fractor .set 000001dH ;
UMIN .set 000001eH ;
UMAX .set 000001fH ;
INA .set 0000020H ;
INB .set 0000021H ;
INC .set 0000022H ;
iq_int .set 0000023H ; running total of iq_integral
id_int .set 0000024H ; running total of id_integral
iqq .set 0000025H ; difference between iqe* and iqe
idd .set 0000026H ; difference between ide* and ide
T_2 .set 0000027H ; tau/2 for use in integrating
tms_iqe .set 0000028H ; commanded value iqe*
tms_ide .set 0000029H ; commanded value ide*
Kpq .set 000002aH ; constant for closed loop
Kiq .set 000002bH ; constant for closed loop
Kpd .set 000002cH ; constant for closed loop
Kid .set 000002dH ; constant for closed loop
*c_v1 .set 000002eH ;
*d_i .set 000002fH ; DC input
*d_v .set 0000030H ;
*x_i .set 0000031H ; Xtra v and i ADC
*x_v .set 0000032H ;
vperfreq .set 0000033H ; Volt per frequency ratio
stopfreq .set 0000034H ; Target frequency
stopvolt .set 0000035H ; Target voltage
tms_invdv .set 0000036H ;
tms_dtset .set 0000037H ;
dmax .set 0000038H ;
dmin .set 0000039H ;
tms_tboost .set 000003aH ;
tms_acscale .set 000003bH ;
tms_dcscale .set 000003cH ;
tms_intbits .set 000003dH ;
tms_outputb .set 000003eH ;
tms_ilmin .set 000003fH ;
tblsize .set 000001aH ; Setup table size
*
*
ports .word 0804500H ; Pointer for i/o ports
ct_swfreq .word 0804000H ; Switching frequency timer
ct_port .word 0804100H ; Timer control register
ct_phasea .word 0804200H ; Phase A timer
ct_phaseb .word 0804300H ; Phase B timer
ct_phasec .word 0804400H ; Phase C timer
d_output .word 0804500H ; General purpuse digital output port
d_input .word 0804600H ; General purpuse digital input port
dac_1 .word 0804700H ; Digital to Analog converter 1
dac_2 .word 0804800H ; Digital to Analog converter 2
inputcs .word 0804900H ; Input voltage and current ADC
acs .word 0804a00H ; Phase a output voltage and current ADC
bcs .word 0804b00H ; Phase b output voltage and current ADC

```



```

ccs .word 0804c00H ; Phase c output voltage and current ADC
adc1_cs .word 0804d00H ; ADC 1
adc2_cs .word 0804e00H ; ADC 2
*
cmd_ad .int startcmd ;
mode_ad .int mode_cmd ;
*
mask_int0 .set 0000001H ; Set external interrupt 0
mask_int1 .set 0000002H ; Set external interrupt 1
mask_int2 .set 0000004H ; Set external interrupt 2
mask_int3 .set 0000008H ; Set external interrupt 3
mask_timer0 .set 0000100H ; Set internal timer 0 interrupt
mask_timer1 .set 0000200H ; Set internal timer 1 interrupt
*
*
clear_main .word 0004444H ;
reset_out .word 000ffffH ;
allon .set 0000000H ;
a_on .set 0000000H ;
a_a3 .set 0000001H ;
a_a4 .set 0000002H ;
b_on .set 0000000H ;
b_a3 .set 0000004H ;
b_a4 .set 0000008H ;
c_on .set 0000000H ;
c_a3 .set 0000010H ;
c_a4 .set 0000020H ;
* Define constants
*
one_pi .float 3.14159263590 ;
two_pi .float 6.28318530718 ;
swp_const .word 10000000 ;
sqrt2 .float 1.414213562373 ;
sqrt3 .float 1.73205080757 ;
sqrt3_3 .float 0.57735026919 ;
sqrt23_3 .float 1.15470053838 ;
half .float 0.5 ;
half12 .float 2048 ;
inv11bits .float 0.00048828125 ;
tenu .float 0.00001 ;
mil .float 0.0001 ;
bi .float 0.2 ;
umax .float 0.05 ;
umin .float -0.05 ;
acdconst .float 0.009765625 ;
acdchalf .float 0.00048828125 ;
acdcmx .float 1.0 ;
acdcmn .float -1.0 ;
zero .float 0.0 ;
ave .float 0.2 ;

tbmax .set 40 ;
tbmin .set 10 ;
*

```

```

cmd      .usect "dualport",10000h
ctio     .usect "xbus",2000h
lookup   .usect "ram1",400h
variable .usect "ram2",400h
        .text

*
*
* ST -- CPU status register
* IE -- CPU/DMA interrupt enable flags
* IF -- CPU interrupt flags
* IOF -- I/O flags
*
* The status register has the following arrangement:
* Bits: 31-14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
* Function: Resrv GIE CC CE CF Res. RM OVM LUF LV UF N Z V C
*
*
* R0:
* R1:
* R2:
* R3:
* R4:
* R5: Saved during interrupt 0,1,2
* R6: Saved during interrupts 0,1,2
* R7: Saved during interrupts 0,1,2
*
* AR0:
* AR1:
* AR2:
* AR3: POINTER FOR INTERNAL MEMORY BLOCK 1 (do not change)
* AR4: POINTER FOR DUAL-PORT MEMORY (do not change)
* AR5: POINTER FOR INTERNAL MEMORY BLOCK 0 (do not change)
* AR6: POINTER FOR SINEWAVE LOOKUP TABLE (do not change)
* AR7: POINTER FOR SINEWAVE LOOKUP TABLE (do not change)
*
* IR0:
* IR1: Saved during interrupts 0,1,2
*
init:    LDI 0,DP      ; Point the DP register to page 0
        LDI 00H,ST     ; Clear and enable cache, and disable OVM (1800h)
        LDI 0000H,IE   ; Clear all interrupts
        LDI @ctrl,AR0   ; Load peripheral bus memory-mapped reg
        LDI @xbus,R0    ;
        STI R0,*+AR0(60H) ; Init expansion bus control reg
        LDI @pbus,R0    ;
        STI R0,*+AR0(64H) ; Init primary bus control reg
        LDI @stck,SP    ; Initialize the stack pointer
        CALL init_ct    ; Init counter/timer
        LDI @d_output,AR0 ;
        LDI 00FFH,R0    ;
        STI R0,*AR0     ;
        LDI @ct_port,AR0 ; Pointer for counter/timer control register
        LDI @reset_out,R0 ;
        STI R0,*AR0     ; Disable all output

```

```

    LDI @ctrl,AR0    ;
*
*
*
    LDI @blk1,AR3    ; Srtach pad memory area
    LDI @dp_mem,AR4   ; Top of dual port memory
    LDI @blk0,AR5     ; Srtach pad memory area
    LDI @sram,AR7     ; Top of the look up table
    LDI @dp_cint,IR0  ; Clear dual port memory interrupt
    LDI *+AR4(IR0),R0 ;
    LDI 000H,R0       ; Clear sram memory
    RPTS 2047         ;
    STI R0,*AR4++(1)  ;
    LDI @dp_mem,AR4   ; Top of dual port memory
    LDI 0000H,IF      ; Clear all flags
    LDI 0008H,IE      ; Enable interrupt 3 (dual port memory)
    OR 02000H,ST      ; Global interrupt enable
*
*
begin: NOP
      NOP
      NOP
      BR begin      ;
*
*
read_cmd: LDI *+AR4(1),R0 ; Check command
          AND 00FFH,R0    ; Clear all other bits
          CMPI 01EH,R0    ;
          BHS stopinit    ; Ignore command if command >= 23
          LDI @cmd_ad,R1  ;
          ADDI R1,R0      ;
          BNZ R0          ;
stopinit: RETS          ;
*
startcmd: BR cmd0        ; Off
          BR cmd1        ; Test Mode ARCP CONTROL
          BR cmd0        ; AC to DC control
          BR cmd0        ; Motor Control - Forward
          BR cmd0        ; Motor Control - Reverse
          BR cmd0        ; Actuator Control - Open
          BR cmd0        ; Actuator Control - Close
          BR cmd0        ; Actuator Control - Open
          BR cmd0        ; Actuator Control - Close
          BR cmd0        ; DC to DC Boost
          BR cmd0        ; AC to DC Control
          BR cmd0        ;
          BR cmd0        ;
          BR cmd0        ;
          BR cmd0        ;
          BR cmd0        ;
          BR cmd0        ;
          BR cmd0        ;
          BR cmd0        ;
          BR cmd0        ;
          BR cmd0        ;

```

```

BR cmd0      ;
BR cmd0      ;
BR cmd0      ;
BR cmd0      ;
BR cmd0      ;
BR cmd0      ;
BR cmd0      ; Stepx
BR cmd0      ; AC output voltage
BR cmd0      ; Boost time
BR cmd0      ; Set current boost limit
BR cmd0      ;
BR cmd0      ;
RETS

*
* Turning off ARCP
*
cmd0:  LDI 08H,IE      ; Disable interrupts 0,1,2
      LDI @ct_port,AR0 ; Pointer for counter/timer control register
      LDI @clear_main,R0 ;
      STI R0,*AR0      ; Disable all output
      STI R0,*+AR3(tms_outputb);
      LDI 030H,R0      ;
wait20: SUBI 01H,R0      ;
      BNZ wait20        ;
      LDI @reset_out,R0 ;
      STI R0,*AR0      ; Disable all counter/timer output
      CALL init_ct      ;
      LDI 00H,R0      ;
      STI R0,*+AR4(1)   ;
      LDI @dp_int,IR0   ;
      STI R0,*+AR4(IR0) ;
      LDI @d_output,AR0 ;
      LDI 0FFFH,R0      ;
      STI R0,*AR0      ;
      LDI @ct_port,AR0 ; Pointer for counter/timer control register
      LDI @reset_out,R0 ;
      STI R0,*AR0      ; Disable all output
      RETS              ;

*
* Test Mode
*
cmd1:  LDI 08H,IE      ; Disable interrupts 0,1,2
      LDI @ct_port,AR0 ; Pointer for counter/timer control register
      LDI @clear_main,R0 ;
      STI R0,*AR0      ; Disable all output
      LDI 030H,R0      ;
wait11: SUBI 01H,R0      ;
      BNZ wait11        ;
      LDI @reset_out,R0 ;
      STI R0,*AR0      ; Disable all counter/timer output

*
      CALL init_ct      ;
      LDI @sram,AR7     ; Reset pointer for phase a
      CALL save_setup   ; Save data in 32-bit format

```

```

CALL set_oc      ;
CALL init_swct   ; Init switching frequency counters
CALL sine_tbl    ; Generate a SINE lookup table
LDI  *+AR3(tms_acv),R0;
FLOAT R0        ;
MPYF @sqrt2,R0   ;
RND  R0          ;
STF  R0,*+AR3(tms_acv);

*
LDF  *+AR3(tms_invdv),R0;
CALL FPINV      ;
LDI  *+AR3(tms_swp),R1;
FLOAT R1        ;
LDF  @half,R2
MPYF3 R1,R2,R3  ;R3=T/2
RND  R3
STF  R3,*+AR3(T_2) ;store T/2 for use in integrating
MPYF R1,R0      ;
*
MPYF @half,R0   ;
RND  R0        ;
STF  R0,*+AR3(tms_dtset);
LDF  *+AR3(tms_tb),R0;
STF  R0,*+AR3(dmax) ;
NEGF R0        ;
STF  R0,*+AR3(dmin) ;
LDF  0.5,R0    ;
STF  R0,*+AR3(INA) ;
STF  R0,*+AR3(INB) ;
STF  R0,*+AR3(INC) ;

LDF  @zero,R0   ;
STF  R0,*+AR3(iq_int);
STF  R0,*+AR3(id_int);
STF  R0,*+AR3(iqq) ;
STF  R0,*+AR2(idd) ;initialize these to zero
*
LDI  *+AR3(tms_acs),R0;
FLOAT R0        ;
MPYF @inv11bits,R0 ;
RND  R0          ;
STF  R0,*+AR3(tms_acscale);
LDI  *+AR3(tms_dcs),R0;
FLOAT R0        ;
MPYF @inv11bits,R0 ;
RND  R0          ;
STF  R0,*+AR3(tms_dcscalc);
*
LDI  *+AR3(tms_blk),R0;
LDI  04H,R1     ;
CALL divi      ;
LDI  R0,R1      ;
STI  R0,*+AR3(tms_cos); Reset pointer for cos function
LDI  *+AR3(tms_blk),BK;
*

```

```

*
    LDI @ctrl,AR0 ; Load peripheral bus memory-mapped reg
    LDI @tim1prd,R0 ; 10ms
    STI R0,*+AR0(38H) ;
    LDI @tim1ctl,R0 ;
    STI R0,*+AR0(30H) ; Init internal timer 1
    LDI @d_output,AR0 ;
    LDI 00FEH,R0 ;
    STI R0,*AR0 ;
    LDI @wait4t,R0 ;
wait31: SUBI 01H,R0 ;
    BNZ wait31 ;
*
    LDI @ct_port,AR0 ; Pointer for counter/timer control register
    LDI allon,R0 ;
    STI R0,*AR0 ; Enable all counter/timer output
    STI R0,*+AR3(tms_outputb);
    LDI 0209H,IE ; Enable interrupts 0,3,9
    LDI 01H,R0 ;
    STI R0,*+AR4(1) ;
    LDI @dp_int,IR0 ;
    STI R0,*+AR4(IR0) ;
    RETS ;
*
*
*
* Initialize counter/timer
*
init_ct: LDI @ct_port,AR0 ; Pointer for counter/timer control register
    LDI 00ffH,R0 ;
    STI R0,*AR0 ; Disable all counter/timer output
    LDI @ct_swfreg,AR0 ; Pointer for switching frequency timer 1
    LDI 0034H,R0 ; Mode 2 (rate generator), 00110100B
    STI R0,*+AR0(3) ;
    LDI 0074H,R0 ; 01110100B
    STI R0,*+AR0(3) ;
    LDI 00b4H,R0 ; 10110100B
    STI R0,*+AR0(3) ;
    LDI @ct_phasea,AR0 ; Pointer for phase a counter
    LDI 0012H,R0 ; Mode 1 (hardware retriggerable one-shoot), 00010010B
    STI R0,*+AR0(3) ;
    LDI 0052H,R0 ; Mode 1, R/W LSB, 01010010B
    STI R0,*+AR0(3) ;
    LDI 00b2H,R0 ; Mode 1, R/W LSB & MSB, 10110010B
    STI R0,*+AR0(3) ;
    LDI @ct_phaseb,AR0 ; Pointer for phase b counter
    LDI 0012H,R0 ; Mode 1 (hardware retriggerable), R/W LSB, 00010010B
    STI R0,*+AR0(3) ;
    LDI 0052H,R0 ; Mode 1, R/W LSB, 01010010B
    STI R0,*+AR0(3) ;
    LDI 00b2H,R0 ; Mode 1, R/W LSB & MSB, 10110010B
    STI R0,*+AR0(3) ;
    LDI @ct_phasec,AR0 ; Pointer for phase c counter
    LDI 0012H,R0 ; Mode 1 (hardware retriggerable), R/W LSB, 00010010B

```

```

    STI R0,*+AR0(3)    ;
    LDI 0052H,R0      ; Mode 1, R/W LSB, 01010010B
    STI R0,*+AR0(3)    ;
    LDI 00b2H,R0      ; Mode 1, R/W LSB & MSB, 10110010B
    STI R0,*+AR0(3)    ;
    RETS

*
save_setup: LDI tblsize,RC    ; Init loop counter
            RPTB save_dp      ;
            LDI *AR4++(1),R0  ; Start at the top of the dual port memory
            AND 0ffH,R0       ; Mask out all higher bits
            LSH 08H,R0        ; Rotate 8 bits to the left
            LDI *AR4++(1),R1  ; Get LSB
            AND 0ffH,R1       ;
            OR  R0,R1         ;
save_dp: STI R1,*AR3++(1)    ; Save 32-bit data in internal RAM
            LDI @dp_mem,AR4    ; Reset AR4
            LDI @blk1,AR3     ; Reset AR3

*
            LDI *+AR3(tms_swf),R1;
*   BZ  init                ; Reset if switching frequency is 0
            LDI @swp_const,R0  ; Determine switching period
            CALL divi          ;
            STI R0,*+AR3(tms_swp);
            LDI 003H,R1        ;
            CALL divi          ;
            MPYI 02H,R0        ;
            ADDI 10H,R0        ;
            STI R0,*+AR3(tms_swp_120)
            LDI *+AR3(tms_swp),R0;
            LDI R0,R1          ; Determine ta
            LSH -1H,R0         ;
*   LDI *+AR3(tms_btime),R2;
*   SUBI R2,R0               ;
            STI R0,*+AR3(tms_ta);
            LDI *+AR3(tms_dt),R0; Determine tb
            LSH 01H,R0         ;
            SUBI R0,R1         ;
            LSH -1H,R1         ;
            FLOAT R1           ;
            RND R1             ;
            STF R1,*+AR3(tms_tb);
            LDI *+AR3(tms_of),R0; Determine stepx
            LDI *+AR3(tms_blk),R1;
            MPYI R1,R0         ;
*   BZ  init                ;
            LDI *+AR3(tms_swf),R1;
            CALL divi          ;
            STI R0,*+AR3(tms_stepx);
            LDI *+AR3(tms_btime),R1
            STI R1,*+AR3(tms_tboost);
            LDI *+AR3(tms_oaci),R2;
            FLOAT R2           ;
            STF R2,*+AR3(tms_ilmin);

```

```

        RETS        ;
*
*
* set overcurrent reference values
*
set_oc: LDI 0FFFH,R0    ;
        MPYI *+AR3(tms_oaci),R0;
        LDI *+AR3(tms_acs),R1;
        CALL divi      ;
        LDI @dac_1,AR0  ;
        STI R0,*AR0    ;
        LDI 0FFFH,R0    ;
        MPYI *+AR3(tms_odci),R0;
        LDI *+AR3(tms_dcs),R1;
        CALL divi      ;
        LDI @dac_2,AR0  ;
        STI R0,*AR0    ;
        RETS
*
*
*
* sine_tbl: this routine generates a SINE lookup table with length equals
*           to the value stored in dp_blk memory location
*
sine_tbl: LDI *+AR3(tms_blk),RC; Get size of lookup table
          LDI RC,R0      ;
          SUBI 0001H,RC  ;
*   BLS init      ; Reset if size is too small
          FLOAT R0      ;
          CALL FPINV     ; 1/blk
          LDF @two_pi,R1 ; Store 2*pi value
          MPYF R1,R0     ; 1/blk * 2 * pi
          RND R0         ;
          LDF R0,R6      ; Save the result
          LDF 0.0,R7     ;
          RPTB save_tbl ;
          MPYF3 R6,R7,R0 ; 1/blk * count * 2 * pi
          CALL SIN       ;
          RND R0         ;
          ADDF 1.0,R7    ; Increment count
save_tbl: STF R0,*AR7++(1) ; Save data into lookup table
          LDI @sram,AR7  ; Restore lookup table pointer
          RETS          ;
*
*
init_swct: LDI @ct_swfreg,AR0 ; Pointer for switching frequency timer 1
          LDI *+AR3(tms_swp),R0;
          STI R0,*+AR0(0) ; Store LSB of counter 0
          LSH -08H,R0     ;
          STI R0,*+AR0(0) ; Store MSB of counter 0
          NOP
          NOP
          NOP
          LDI *+AR3(tms_swp_120),R2;

```



```

checkout0:LDI 0000H,R0      ;
          STI R0,*+AR0(3)   ; Latch command
          LDI *+AR0(0),R0   ;
          AND 000FFH,R0     ; Clear all other higher bits
          LDI *+AR0(0),R1   ;
          LSH 0008H,R1      ;
          AND 00f00H,R1     ;
          OR  R1,R0         ;
          CMPI R2,R0        ;
          BGT checkout0    ;
          LDI *+AR3(tms_swp),R0;
          STI R0,*+AR0(1)   ; Store LSB of counter 1
          LSH -0008H,R0     ;
          STI R0,*+AR0(1)   ; Store MSB of counter 1
          NOP
          NOP
          NOP
          LDI *+AR3(tms_swp_120),R2;
checkout1:LDI 0040H,R0      ;
          STI R0,*+AR0(3)   ; Latch command
          LDI *+AR0(1),R0   ;
          AND 000FFH,R0     ; Clear all other higher bits
          LDI *+AR0(1),R1   ;
          LSH 0008H,R1      ;
          AND 00f00H,R1     ;
          OR  R1,R0         ;
          CMPI R2,R0        ;
          BGT checkout1    ;
          LDI *+AR3(tms_swp),R0;
          STI R0,*+AR0(2)   ; Store LSB of counter 2
          LSH -0008H,R0     ;
          STI R0,*+AR0(2)   ; Store MSB of counter 2
*
          LDI @ct_phasea,AR0 ; Pointer for phase a counter
          LDI *+AR3(tms_btime),R1 ;
          STI R1,*+AR0(0)   ; Store LSB of counter 0
          LDI *+AR3(tms_bdly),R1 ;
          ADDI *+AR3(tms_btime),R1 ;
          STI R1,*+AR0(1)   ; Store LSB of counter 1
          LDI *+AR3(tms_ta),R1 ;
          STI R1,*+AR0(2)   ; Store LSB of counter 2
          LSH -08H,R1       ;
          STI R1,*+AR0(2)   ; Store MSB of counter 2
*
          LDI @ct_phaseb,AR0 ; Pointer for phase b counter
          LDI *+AR3(tms_btime),R1 ;
          STI R1,*+AR0(0)   ; Store LSB of counter 0
          LDI *+AR3(tms_bdly),R1 ;
          ADDI *+AR3(tms_btime),R1 ;
          STI R1,*+AR0(1)   ; Store LSB of counter 1
          LDI *+AR3(tms_ta),R1 ;
          STI R1,*+AR0(2)   ; Store LSB of counter 2
          LSH -08H,R1       ;
          STI R1,*+AR0(2)   ; Store MSB of counter 2

```

```

*
    LDI @ct_phasec,AR0 ; Pointer for phase c counter
    LDI *+AR3(tms_btime),R1 ;
    STI R1,*+AR0(0) ; Store LSB of counter 0
    LDI *+AR3(tms_bdly),R1 ;
    ADDI *+AR3(tms_btime),R1 ;
    STI R1,*+AR0(1) ; Store LSB of counter 1
    LDI *+AR3(tms_ta),R1 ;
    STI R1,*+AR0(2) ; Store LSB of counter 2
    LSH -08H,R1 ;
    STI R1,*+AR0(2) ; Store MSB of counter 2
    LDI *+AR3(tms_btime),R0 ;
    STI R0,*+AR3(tms_tboost);
    RETS

*
*
isr_mode: LDI *+AR3(tms_mode),R0 ;
    LDI @mode_ad,R1 ;
    ADDI R1,R0 ;
    BNZ R0 ;
    RETS ;

*
mode_cmd: BR mode0 ; Stop
    BR mode1 ; Test Mode (ARCP Open-loop)
    BR mode0 ; DC to AC Mode
    BR mode0 ; Motor Control Mode - Forward
    BR mode0 ; Motor Control Mode - Reverse
    BR mode0 ; Actuator Control Mode - Open
    BR mode0 ; Actuator Control Mode - Close
    BR mode0 ; Linear Actuator Mode - Open
    BR mode0 ; Linear Actuator Mode - Close
    BR mode0 ; DC to DC Boost Mode
    BR mode0 ; DC to DC Buck Mode
    BR mode0 ; Stop
    BR mode0 ; Stop

*
mode0: LDI 08H,IE ; Disable interrupts 0,1,2
    LDI @ct_port,AR0 ; Pointer for counter/timer control register
    LDI @clear_main,R0 ;
    STI R0,*AR0 ; Disable all output
    LDI 030H,R0 ;
wait30: SUBI 01H,R0 ;
    BNZ wait30 ;
    LDI @reset_out,R0 ;
    STI R0,*AR0 ; Disable all counter/timer output
    AND 08H,IF ; Clear all pending interrupts 0,1,2
    LDI 00H,R0 ;
    STI R0,*+AR4(1) ;
    LDI @dp_int,IR0 ;
    STI R0,*+AR4(IR0) ;
    RETS ; Return

*
*
mode1: MPYF *+AR3(tms_tb),R7;

```

```

        FIX R7          ;
        ADDI *+AR3(tms_ta),R7;
        RETS           ; Return
*
* timer0: Motor startup timer
*
time0:  PUSH R0          ;
        PUSHF          R0          ;
        PUSH AR0        ;
        LDI *+AR3(tms_stepx),R0;
        ADDI 01H,R0      ;
        STI R0,*+AR3(tms_stepx);
        CMPI *+AR3(stopfreq),R0;
        BLT looptimer0   ;
        LDI 000H,R0      ;
        LDI @ctrl,AR0    ;
        STI R0,*+AR0(20H) ; Clear counter
        ANDN mask_timer0,IE ; Disable timer interrupt
        POP AR0          ;
        POPF R0          ;
        POP R0           ;
        RETI            ;
looptimer0: LDF *+AR3(vperfreq),R0;
        ADDF *+AR3(tms_acv),R0;
        RND R0           ;
        STF R0,*+AR3(tms_acv);
        POP AR0          ;
        POPF R0          ;
        POP R0           ;
        RETI            ;
*
* timer1: Discharging circuit
*
time1:  PUSH R0          ;
        PUSHF          R0          ;
        PUSH AR0        ;
        LDI @d_output,AR0 ;
        LDI 0FFH,R0      ;
        STI R0,*AR0      ;
        LDI 000H,R0      ;
        LDI @ctrl,AR0    ;
        STI R0,*+AR0(30H) ; Clear counter
        ANDN mask_timer1,IE ; Disable timer interrupt
        POP AR0          ;
        POPF R0          ;
        POP R0           ;
        RETI            ;
*
*
* irs0: Phase A interrupt service routine
**irs0 for closed-loop control of ARCP resonant converter
*
*written by David Floodeen
*
```

```

isr0: PUSH    ST          ; Save registers
      PUSH    IR1        ;
      PUSH    R7         ;
      PUSHF   R7         ;
      PUSH    R6         ;
      PUSHF   R6         ;
      PUSH    R5         ;
      PUSHF   R5         ;
      PUSH    R4         ;
      PUSHF   R4         ;
      PUSH    R3         ;
      PUSHF   R3         ;
      PUSH    R2         ;
      PUSHF   R2         ;
      PUSH    R1         ;
      PUSHF   R1         ;
      PUSH    R0         ;
      PUSHF   R0         ;
      PUSH    AR0        ;
*
      LDI     @acs,AR0    ; Pointer for phase a A/D converter
      LDI     @bcs,AR2    ; Pointer for phase b A/D converter
      LDI     *AR0,R0     ; initiate a new conversion (don't use these values, they are time-late)
      LDI     *AR2,R1     ;
      LDI     00cH,R2     ; delay loop to allow time for the slower A/D converters
wait: SUBI    01H,R2
      BNZ     wait
*
* READ and STORE SAMPLED CURRENTS
*
      LDI     *AR0,R0     ; READ Va AND ia
      LDI     *AR2,R2     ; Read Vb and ib

***Get ia and ib... Assuming ia is msb of A/D word as in Tuan's code
      LSH     04H,R0      ;
      ASH     -14H,R0     ;
      FLOAT   R0          ; R0 = ia

      LSH     04H,R2      ;
      ASH     -14H,R2     ;
      FLOAT   R2          ; R2 = ib

*****
* This section scales the A/D current using a scaling factor to make them
* actual currents
* Final output of this section R0 = ia, R2 = ib

***assuming tms_acscale is set to the current scaling word calculated by Ron Hanson***

      MPYF    *+AR3(tms_acscale),R0
      RND     R0          ;R0 = ia
*      STF     R0,*+AR3(ia) ; STORE ia

```

```

        MPYF  *+AR3(tms_acscale),R2
        RND   R2                      ;R2 = ib
*       STF   R2,*+AR3(ib)            ; STORE ib

```

*This section reads sin_theta and cos_theta from the lookup table

***assuming tms_cos is block size / 4 vice tms_13 or tms_23

```

        LDI   *+AR3(tms_stepx),IR1    ;
        LDF   *AR7++(IR1)%,R6         ;R6 = sin_theta

        LDI   *+AR3(tms_cos),IR1      ;
        SUBI  *+AR3(tms_stepx),IR1    ;subtract step to compensate for previous increment
        LDI   AR7,AR6                 ;use AR6 to not further increment AR7
        LDF   *AR6++(IR1)%,R7         ;read twice to get desired value in R7
        LDF   *AR6++(IR1)%,R7         ;R7 = cos_theta

```

*This section converts ia and ib to iqs and ids then to iqe and ide

***assumes sqrt3_3 = (root3)/3, sqrt23_3 = (2root3)/3

```

        LDF   R0,R1                   ;R0 = iqs = ia = R1

        MPYF  @sqrt3_3,R1             ;R1=(root3)/3*ia
        MPYF  @sqrt23_3,R2            ;R2=(2root3)/3*ib
        SUBF  R2,R1                   ;R1 = ids = (root3/3)ia-(2root3/3)ib

        MPYF3 R7,R0,R2                ;R2=iqs(cos_theta)
        MPYF3 R6,R1,R3                ;R3=ids(sin_theta)
        SUBF  R3,R2                   ;R2=iqe=iqs(cos_theta)-ids(sin_theta)

        MPYF3 R6,R0,R3                ;R3=iqs(sin_theta)
        MPYF3 R7,R1,R4                ;R4=ids(cos_theta)
        ADDF  R4,R3                   ;R3=ide=iqs(sin_theta)+ids(cos_theta)

```

*This section calculates iqq and idd

```

        LDF   *+AR3(tms_iqe),R0       ;
        SUBF  R2,R0                   ;R0=iqq=iqe* - iqe
        LDF   *+AR3(tms_ide),R1      ;
        SUBF  R3,R1                   ;R1=idd=ide* - ide

```

*Integrate using trapazoidal method to calculate iq_int and id_int

* and calculates Vqe and Vde

***assuming iq_int, id_int, iqq, idd initialized = 0

***assuming T_2 = T/2 is calculated already

```

        LDF   R0,R2                   ;R0=R2=iqq
        ADDF  *+AR3(iqq),R2           ;R2=iqq[n-1]+iqq[n]
        STF   R0,*+AR3(iqq)          ;store iqq for next time
        MPYF  *+AR3(T_2),R2          ;R2=T/2(iqq[n-1]+iqq[n])
        ADDF  *+AR3(iq_int),R2       ;R2=iq_int[n-1]+T/2(iqq[n-1]+iqq[n])

```

```

STF    R2,*+AR3(iq_int)      ;store iq_int for next time
MPYF   *+AR3(Kiq),R2 ;
MPYF   *+AR3(Kpq),R0 ;
ADDF   R0,R2                  ;R2=Vqe

LDF    R1,R3                  ;R1=R3=idd
ADDF   *+AR3(idd),R3 ;R3=idd[n-1]+idd[n]
STF    R1,*+AR3(idd) ;store idd for next time
MPYF   *+AR3(T_2),R3 ;R3=T/2(idd[n-1]+idd[n])
ADDF   *+AR3(id_int),R3      ;R3=id_int[n-1]+T/2(idd[n-1]+idd[n])
STF    R3,*+AR3(id_int)      ;store id_int for next time
MPYF   *+AR3(Kid),R3 ;
MPYF   *+AR3(Kpd),R1 ;
ADDF   R1,R3                  ;R3=Vde

```

*This section transforms Vqe and Vde to Vqs and Vds

```

MPYF3  R2,R7,R0      ;R0=Vqe*cos theta
MPYF3  R3,R6,R1      ;R1=Vde*sin theta
ADDF   R1,R0          ;R0=Vqs

MPYF3  R3,R7,R1      ;R1=Vde*cos theta
MPYF3  R2,R6,R4      ;R4=Vqe*sin theta
SUBF   R4,R1          ;R1=Vds

```

*This section transforms Vqs and Vds to Va, Vb, Vc

***assumes @half=.5, @sqrt3=1.7320508, @zero=0.0

```

LDF    R0,R3          ;Vqs=R0 = R3=Va

MPYF   @half,R0      ;R0=.5Vqs
MPYF   @half,R1      ;R1=.5Vds
MPYF   @sqrt3,R1     ;R1=(root3)Vds/2
LDF    @zero,R2      ;R2=0.0
SUBF3  R0,R2,R4      ;R4= -.5Vqs
SUBF   R1,R4          ;R4= -.5Vqs-(root3)Vds/2 = Vb

SUBF3  R3,R2,R5      ;R5= -Va
SUBF   R4,R5          ;R5= -Va-Vb = Vc

```

*This section calculates new duty_counta, b, c

```

MPYF   R6,R3          ;R3=Va*sin theta
ADDF   *+AR3(tms_ta),R3 ;
FIX    R3             ;R3=dutycount_a

MPYF   R6,R4          ;R4=Vb*sin theta
ADDF   *+AR3(tms_ta),R4 ;
FIX    R4             ;R4=dutycount_b

MPYF   R6,R5          ;R5=Vc*sin theta
ADDF   *+AR3(tms_ta),R5 ;
FIX    R5             ;R5=dutycount_c

```

*This section loads new duty_counta, b, c

```

    LDI    @ct_phasea,AR0
    STI    R3,*+AR0(2)        ;stores lsb of counter
    LSH    -08H,R3
    STI    R3,*+AR0(2)        ;stores msb of counter

    LDI    @ct_phaseb,AR0
    STI    R4,*+AR0(2)        ;stores lsb of counter
    LSH    -08H,R4
    STI    R4,*+AR0(2)        ;stores msb of counter

    LDI    @ct_phasec,AR0
    STI    R5,*+AR0(2)        ;stores lsb of counter
    LSH    -08H,R5
    STI    R5,*+AR0(2)        ;stores msb of counter

```

*This section clears the interrupt and the stack

```

    ANDN   mask_int0,IF    ; Clear interrupt 0
*
    POP    AR0              ;
    POPF   R0               ;
    POP    R0               ;
    POPF   R1               ;
    POP    R1               ;
    POPF   R2               ;
    POP    R2               ;
    POPF   R3               ;
    POP    R3               ;
    POPF   R4               ;
    POP    R4               ;
    POPF   R5               ;
    POP    R5               ;
    POPF   R6               ;
    POP    R6               ;
    POPF   R7               ;
    POP    R7               ;
    POP    IR1              ;
    POP    ST               ;
*
    RETI                    ; Return and enable interrupt
*

```

* isr1: Phase B interrupt service routine

```

*
isr1: NOP                  ;
    RETI                    ; Return interrupt not used
*

```

* isr2: Phase C interrupt service routine

```

isr2: NOP
    RETI                    ; Return interrupt not used
*

```

* irs3: Dual port memory interrupt service routine

```

*
isr3:  PUSH ST      ; Save registers
        PUSH DP      ;
        PUSH IR1     ;
        PUSH R7      ;
        PUSHF R7     ;
        PUSH R6      ;
        PUSHF R6     ;
        PUSH R5      ;
        PUSHF R5     ;
        PUSH R4      ;
        PUSHF R4     ;
        PUSH R3      ;
        PUSHF R3     ;
        PUSH R2      ;
        PUSHF R2     ;
        PUSH R1      ;
        PUSHF R1     ;
        PUSH R0      ;
        PUSHF R0     ;

*
        LDI @dp_cint,IR0 ;
        LDI *+AR4(IR0),R0 ; Clear interrupt
        CALL read_cmd    ;
        ANDN mask_int3,IF ; Clear interrupt 3

*
        POPF R0      ;
        POP R0       ;
        POPF R1      ;
        POP R1       ;
        POPF R2      ;
        POP R2       ;
        POPF R3      ;
        POP R3       ;
        POPF R4      ;
        POP R4       ;
        POPF R5      ;
        POP R5       ;
        POPF R6      ;
        POP R6       ;
        POPF R7      ;
        POP R7       ;
        POP IR1      ;
        POP DP       ;
        POP ST       ;

*
        RETI         ; Return and enable interrupt

*
*
        .end

```


LIST OF REFERENCES

1. Dade, T.B., "Advanced Electric Propulsion, Power Generation, and Power Distribution," *Naval Engineers Journal*, Vol. 106, No. 2, pp. 83-92, March, 1994.
2. Oberley, M. J., "The Operation and Interaction of the Auxiliary Resonant Commutated Pole Converter in a Shipboard DC Power Distribution Network," Master's Thesis, Naval Postgraduate School, Monterey, CA, 1996.
3. Hanson, R. J., "Implementing Closed-loop Control Algorithms for DC-to-DC Converters and ARCP Inverters Using the Universal Controller," Electrical Engineer Thesis, Naval Postgraduate School, Monterey, CA, June, 1997.
4. Fisher, M. J., *Power Electronics*, PWS-Kent Publishing Company, Boston, 1991.
5. NSWC/CDAD Schematics, Code 813, Annapolis, 1995.
6. Intel, "AP-70 Using the INTEL MCS 51 Boolean Processing Capabilities", Intel Corporation, 1998.
7. MAXIM, "MAX120/MAX122 Data Sheet," Sunnyvale, CA, 1994.
8. Texas Instruments, "TMS320C3x User's Guide," Texas Instruments, Inc., 1994.
9. Texas Instruments, "TMS320 Floating-Point DSP Optimizing Compiler," Texas Instruments, Inc., 1991.
10. Texas Instruments, "TMS320 Floating-Point DSP Assembly Language Tools," Texas Instruments, Inc., 1991.
11. Texas Instruments, "TMS320C3X C Source Debugger," Texas Instruments, Inc., 1993.
12. Deitel H.M., Deitel P.J., *C How to Program*, Prentice-Hall, Inc., 1994.
13. DeDoncker, R.W., Lyons, J.P., "The Auxiliary resonant Commutated Pole Converter," *IEEE-IAS Annual Meeting Proceedings*, 1990 pp. 1228-1235.
14. Mayer, J.S., Salberta, F., "High-Frequency Power Electronic Converter For Propulsion Applications," Final Technical Report, Department of Electrical Engineering and the Applied Research Laboratory, Penn State University, University Park, PA

15. Microsoft, "MS-DOS 6 User's Manual", Microsoft Corporation, 1993.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
8725 John J. Kingman Rd., STE 0944
Fort Belvoir, Virginia 22060-6218
2. Dudley Knox Library.....2
Naval Postgraduate School
411 Dyer Rd.
Monterey, California 93943-5101
3. Chairman, Code EC.....1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5121
4. John Ciezki, Code EC/Cy.....3
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5121
5. Robert Ashton, Code EC/Ah.....3
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5121
6. David L. Floodeen.....2
137 Moreell Circle
Monterey, California 93940